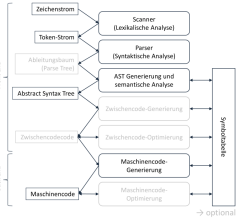
1 Programmiersprachen

Einteilung

- **Deklarative Sprachen**
  - Was soll der Computer tun?
  - Funktionale, logische und datenflussorientierte Sprachen
    - funktional: LISP, Scheme, Scala, Haskell, ...
    - logisch (regelbasiert): Prolog, XSLT, ...
    - domain specific language (DSL): SQL, Excel, YACC, XAML, ...
- **Imperative Sprachen**
  - Wie soll der Computer etwas tun?
  - von-Neumann, Skript- und objektorientierte Sprachen
    - prozedurale (von Neumann): FORTRAN, C, ...
    - objektorientiert: C++, Java, C#, Swift, ...
    - Skriptsprachen (interpretierbar, dyn, typisiert): JavaScript, Python, ...

2 Compile-Phasen



3 Syntaxspecifikation

- Reguläre Ausdrücke -> reguläre Menge
  - Werden durch **scanner**
- Kontextfreie Grammatik (CFG) -> Kontextfreie Sprache (CFL)
  - Werden durch **parser** erkannt

4 Parser & Kontextfreie Grammatik

Parser

- Basiert auf kontextfreier Grammatik (CFG)
- Prüft, ob ein Satz zur Sprache gehört
- Erstellt Ableitungsbaum (Derivationsbaum)
- Laufzeit für CFGs: $O(n^3)$ (z.B. Earley, CYK)
- Effizientere Parser für bestimmte CFGs: LL, LR (Laufzeit $O(n)$)

Kontextfreie Grammatik (CFG)

Beispiel: Reguläre Ausdrücke

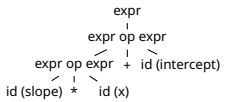
```
number  → integer | real
integer → digit digit*
real    → integer exponent | decimal (exponent | )
decimal → digit* (. digit | digit .) digit*
exponent → (e | E) (+ | - | ) integer
digit   → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Beispiel: Rekursive Definition

```
expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /
```

Ableitung und Derivationsbaum

```
expr → expr op expr
     → expr op id
     → expr + id
     → expr op expr + id
     → expr op id + id
     → id * id + id
```



Parser Typen

LL-Parser

- Links-nach-rechts, links-hergeleitete Ableitungen
- LL(k): k Vorausschau-Tokens
- Einfach zu implementieren (rekursiver Abstieg)

LR-Parser

- Links-nach-rechts, rechts-hergeleitete Ableitungen
- Mehr Ausdrucksstärke als LL-Parser
- Automatisch generierbar, komplexere Zustandstabellen

Fehlerbehandlung

- Aussagekräftige Fehlermeldungen
- Lexikalische Fehler: ungültige Zeichen (Scanner)
- Syntaxfehler: ungültige Token-Reihenfolge (Parser)

5 Semantische Analyse

- untersucht nur statische Bindungen und Regeln
  - Bezeichner / Objekte / Typen
- Erzeugt dabei
  - Symboltabelle
  - Abstrakten Syntaxbaum (AST)

5.1 Symboltabelle

Datenbanken

- Deklaration und Definition m Ableitungsbaum
- Bindung von Namen zu Objekten
- Objektarten und Merkmale
  - Namensraum, Typ, Methode/Funktion, Parameter
  - Variable, Konstanten, Wert, Adresse, Sichtbarkeit,...
- Baumstruktur spiegelt Gültigkeitsbereiche wieer
  - Namen kann aufgelöst (in der Umgebung gefunden) werden
  - Mehrdeutige Namen werden auf eindeutige Symbole abgebildet
    - Sonst Fehler

5.2 Abstract Syntax Tree

Zwischencode als Baumstruktur

- Reduzierte und augmentierte Form des Ableitungsbaum (keine Deklarationen, Satzzeiche, etc...)
- Operationen sind Elternknoten -> Operanden Kindknoten
- Namen werden durch verweis auf Symboltabelle aufgelöst
- Sequenzen werden verkettet Anweisung, Methoden, usw.

Typen im AST

- Objekte in Symboltabelle und literale Konstanten haben einen Typ
- Typen werden im AST propagiert
  - *Typprüfung:* Typfehler erkenne
  - Fehlende Typen in Symboltabelle nachtragen (Typinferenz)
  - Überladene Operationen und literale Konstanten auflösten
  - *Typanpassungen* (implizierte Konversionen) ergänzen
  - Generische Typen instanziieren

5.3 Zwischencode

Mögliche Form der Codedarstellung

- Ableitungsbaum
  - Ermöglicht automatisierte, formatierungstreue Source-to-Source Compiler ("Transpiler")
- AST (Abstrakter Syntaxbaum)
  - Implementierung neuer Sprachkonstruktionen mittels "Lowering"
    - Neue Spracheleme werden durch existierende ersetzt
- Intermediate Language / Representation (IL / IR)
  - Maschinenunabhängige Optimierungen
  - Maschienenunaghängige Comiler-Frontend
  - Erleichterte Protierung auf andere Zielhardware

5.4 Erzeugung von Maschienencode

Code Generierung

- Symboltabelle wird um Speicheradressen ergänzt
  - Teilweise Stack-/Framepointer-relativ
- Erzeugen von Zielcode
  - Assemblercode
  - Maschienencode
- Teilfaugaben
  - Befehlsauswahl
  - Registeralokation
  - Befehlsanordnung

Optimierung

- Maschienenabhängige Verfahren
  - Architekturspezifische Maschienenbefehle oder Adressierungsarten
    - z.B. SSE, AVX, usw...
  - Keyhole-Codeverbesserungen
    - Folgen von Befehlen durch gleichwertige (aber schnellere) ersetzen
  - Chache-Coherenze verbessern
    - Speicherlayout, Zugriffsmuster
  - Nicht-trivial
    - Multi-core, multi-threaded Architekuren

Optimierung nach **Max speed** oder **Min size**

5.5 Compiler Performance

- Schnelle Compiler
  - Vermeiden merfache Läufe über das Program
  - Vermeiden eine Datenstrukturen wie Token-Datai, Ableitungsbaum, AST
  - Übersetzen inkrementell nur geänderten Code
- Übersetzung ist syntaxgesteuert: Parser...
  - ...holt Token vom Scanner
  - ...erzeugt und attributiert AST-Fragment
  - ...aktulaisert Symboltabelle
  - ...sendet Fragmente und Codegenerierung

5.6 Laufzeitsysteme

Linken Laufzeitsysteme

- Das Laufzeitsystem ist fixer Code, der zur Aufzührung zwingend notwenig ist
  - Code-Verifikation
  - Interpretation oder JIT-Übersetzung
  - Exceptin-Handling
  - Garbage-Collectoin
  - Nachladen und Linken zur Laufzeit
- Laufzeitsystem bildet "virtuelle Maschiene"
  - Implementiert Spracheigenschaften, die nicht durch den Compiler abgebildet werden könnn
  - Wird nur indirekt verwedet
  - Eingebettete Software had i.d.R kein oder neu ein minimales Laufzeitsystem
- Bibliotheken sind dagegen immer optional
  - Selbst eine Standardbibilothek

6 Namen und Bindungen

6.1 Namen

- Token: Bezeichner, Literale, Operatoren, Schlüsselwörter
- Beispiele: `foo`, `3.14`, `+`, `while`

6.2 Bindungen

- Namensbindung: Name an Objekt/Wert/Typ
- Typbindung: Typ an Objekt/Wert
- Wertbindung: Wert an Objekt
- Adressbindung: Speicheradresse an Objekt
- Binden ist das Erzeugen einer Zuordnung

6.3 Bedeutung von Namen

- Aliasing: Mehrere Namen für ein Objekt
- Überladung: Ein Name für mehrere Operationen

6.4 Bindungszeitpunkt

- Statische Bindung: zur Compilezeit (z.B. Typ einer Variable)
- Dynamische Bindung: zur Laufzeit (z.B. Wert einer Variable)

6.5 Gültigkeitsbereiche

- Globaler, Modul-, Namensraum-, Datei-, Funktions-, Block-, Ausdrucks-Gultigkeitsbereich
- Verdeckte Bindungen: Lokale Bindung verdeckt ausere Bindung
- Freie Variablen: Variablen ohne lokale Bindung, Bezug auf ausere Bindung

6.6 Closures

- Eine Closure ist eine Funktion mit ihrem Umfeld, die alle freien Variablen bindet.
- Erlaubt Funktionen auf Variablen außerhalb ihres lokalen Gultigkeitsbereichs zuzugreifen.
- Beispiel in JavaScript:

```
1function outer() {
2    let a = 1;
3    function first() {
4        let a = 2;
5        return second();
6    }
7    function second() {
8        return a;
9    }
10   return first();
11}
12// second() greift auf a=1 aus outer() zu
```

- Hier ist `a` in `second` eine freie Variable, die durch `outer` gebunden wird.
- Detecting closures: Identifizieren Sie alle Funktionen und prüfen Sie, ob sie auf Variablen aus einem äußeren Gültigkeitsbereich zugreifen.

6.7 Aliasing

- Zwei oder mehr Namen referenzieren ein Objekt.
- Häufig in Sprachen mit Zeigern und Referenzen.
- Erschwert das Verständnis und verhindert Optimierungen.

6.7.1 Beispiel: C++

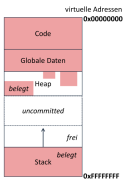
```
1double sum = 0.0;
2void accumulate(double& x) {
3  sum += x;
4  sum_of_squares += x * x;
5}
6accumulate(sum); // sum is x!
```

6.7.2 Beispiel: C#

```
1class Example {
2  int value_ = 42;
3    public int value {
4        get { return value_; }
5        set { value_ = value; }
6    }
7}
8class Program {
9    static void Main() {
10       Example a = new Example();
11       Example b = a;
12       b.value = 10;
13   }
14}
15// a.value_ == b.value_
```

7 Speicher

- **Speichersegmente:**
  - **Code-Segment:** Enthält den ausführbaren Programmcode (read-only).
  - **Datensegment:** Globale und statische Variablen.
  - **Stack:** LIFO-Struktur für Funktionsaufrufe und lokale Variablen.
  - **Heap:** Dynamische Speicherallokation für Objekte.
- **Allokationsmechanismen:**
  - **Statisch:** Feste Adresse zur Compilezeit.
  - **Stack-basiert:** Allokation und Freigabe in LIFO-Reihenfolge.
  - **Heap-basiert:** Beliebige Allokation und Freigabe, komplexere Verwaltung.
- **Lebensdauer von Objekten:**
  - **Stack:** Automatische Allokation bei Funktionsaufruf, Freigabe bei Rückkehr.
  - **Heap:** Manuelle Allokation (z.B. `new`/`malloc`) und Freigabe (`delete`/`free`), oder automatische Freigabe durch Garbage Collection.
- **Speicherprobleme:**
  - **Garbage:** Nicht mehr erreichbarer Speicher, der nicht freigegeben wurde.
  - **Memory Leak:** Allokierter Speicher, der nie freigegeben wird.
  - **Dangling Reference:** Verweis auf bereits freigegebenen Speicher.

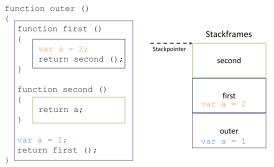


7.1 Gültigkeitsbereiche

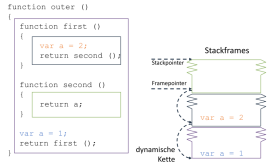
- **Stackframe:**
  - Enthält Variablen im Gültigkeitsbereich einer Funktion/Prozedur.
  - Konstanten und Typen sind im Datensegment.
  - Compiler löst Gültigkeitsbereiche innerhalb einer Prozedur während der semantischen Analyse auf.
- **Activation Record:**
  - Implementierung auf dem Heap erlaubt das Festhalten einer Umgebung.

7.2 Implementierung

- **Bezugspunkt Stackzeiger:**
  - Bei Prozedureintritt/-austritt wird der Stackzeiger um die feste Stackframegröße versetzt.
  - Beispiel in JavaScript:



- **Bezugspunkt Framezeiger:**
  - Vereinfacht Implementierung, wenn die Stackframe-Größe einer Prozedur variieren kann.
  - Bei Prozedureintritt wird der alte Framezeiger gesichert und neu aus dem Stackzeiger berechnet.
  - Beispiel in JavaScript:



  - Alte Framezeiger im Stack bilden eine "dynamische Kette".

7.3 Der Heap

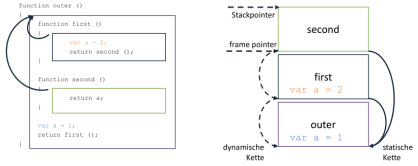
- Dynamische Speicherverwaltung:
  - Belegte und freie Blöcke haben ihre Länge gespeichert.
  - Der Heap kann linear durchlaufen werden, z.B. für Garbage Collection.
  - Freie Blöcke sind oft verkettet, um schneller gefunden zu werden.

7.4 Lebensdauer von Objekten

- **Prinzip:**
  - Lebensdauer darf nicht kürzer sein als Erreichbarkeit (z.B. Gültigkeitsbereich der Namensbindung).
- **Globale Daten:**
  - Globale Lebensdauer, keine weitere (De-)Allokation möglich.
  - Beispiele: Konstanten, globale Variablen, Klassenvariablen, static Variablen, Module.
- **Stack:**
  - Automatische Allokation bei Prozeduraufruf.
  - Freigabe mit Return (Löschen des Stackframes).
- **Heap:**
  - Explizite Allokation mit expliziter Freigabe (`new`/`delete` oder `malloc`/`free`).
  - Explizite Allokation mit automatischer Freigabe (Garbage Collection).
- **Probleme:**
  - **Garbage:** Lebensdauer ohne Erreichbarkeit.
  - **Memory Leak:** Fehlende explizite Freigabe.
  - **Dangling Reference:** Erreichbarkeit nach (expliziter) Freigabe.

8 Geschachtelte Funktionen

- **Lokale Funktionen:**
  - Definiert innerhalb anderer Funktionen, nützlich für Hilfsfunktionen.
  - Erlauben funktionale Programmierung mit geschachtelten (anonymen) Funktionen.
  - Rust erlaubt lokale Funktionen ohne Zugriff auf die äußere Umgebung.
- **Implementierung:**
  - **Statische Kette:** Verweise in Stackframes auf die umgebenden Gültigkeitsbereiche.
  - Compiler/Interpreter platziert Zeiger auf den nächsten lexikalischen Gültigkeitsbereich.
  - Die statische Kette ermöglicht Zugriff auf Variablen in äußeren Gültigkeitsbereichen.



9 Speichersegmente

- **Code-Segment**: Enthält den ausführbaren Code des Programms.
- **Globale Daten**:
  - Beinhaltet globale, statische Variablen, Konstanten und Metadaten.
  - Unterteilungen: Initialisiert, uninitialisiert, genullt, read-write, read-only.
- **Heap**: Dynamische Speicherallokation, keine feste Reihenfolge.
- **Stack**: LIFO-Allokation, wachst abhängig von der Hardware.

10 Allokationsmechanismen

- **Statisch**: Feste Adresse zur Compilezeit.
- **Stack-basiert**: LIFO-Prinzip, hauptsächlich für Funktionen und lokale Variablen.
- **Heap-basiert**: Beliebige Allokation und Zerstörung, komplexes Speichermanagement.

11 Adressierung

- **Konstanten**: Feste Adresse im Segment "Globale Datenöder eingebettet.
- **Globale Daten**: Feste Adresse im Segment "Globale Daten".
- **Lokale Variablen, Parameter**: Fester Offset relativ zum Stackpointer oder Framepointer.

12 Stackframe

- Abhängig von Programmiersprache, Betriebssystem, Aufrufkonvention, Hardware, ABI.
- **Caller** reserviert Platz für Parameter und lädt Argumente in den Stackframe.
- **Callee** verschiebt den Stackpointer, reserviert Platz für lokale Daten und gesicherte Register.

13 Microsoft x86-64 ABI

- Stack wächst von hohen zu niedrigen Adressen, 16-Byte Alignment.
- **call**: Sichert alten RIP und lädt Sprungziel.
- **ret**: Restauriert alten RIP.
- **Parameter/Rückgabewert**: Erste vier Parameter in Registern, größere in Speicher.
- **Register**:
  - Flüchtig: RAX, RCX, RDX, R8-R11.
  - Nicht-Flüchtig: RBX, RBP, RDI, RSI, RSP, R12-R15.

14 System V AMD64 ABI

- **Parameterübergabe**:
  - POINTER/INTEGER: RDI, RSI, RDX, RCX, R8, R9.
  - SSE: XMM0-7 für Floating-Point Parameter.
  - MEMORY: Stack für große Objekte.
- **Flüchtige/Nicht-Flüchtige Register**:
  - Flüchtig: RAX, RDI, RSI, RDX, RCX, R8-R11.
  - Nicht-Flüchtig: RBX, RSP, RBP, R12-R15.
- **Stack Layout**:
  - 16-Byte Alignment, 128-Byte "Red Zone".

15 Registersatz

64bit Register	32bit Register	16bit Register	Bedeutung	volatile (caller saved)	nonvolatile (callee preserved)	Params	Params (large retval)
RIP			instruction pointer		(x)		
RFLAGS			status&control	(x)			
RAX	EAX	AX	general purpose	x		retval	&retval
RCX	ECX	CX	general purpose	x		p1	&retval
RDX	EDX	DX	general purpose	x		p2	p1
RBX	EBX	BX	general purpose		x		
RSP	ESP	SP	stack pointer		x		
RBP	EBP	BP	general purpose (base ptr)		x		
RSI	ESI	SI	general purpose (src index)		x		
RDI	EDI	DI	general purpose (dst index)		x		
R8			general purpose	x		p3	p2
R9			general purpose	x		p4	p3
R10..R11			general purpose	x			
R12..R15			general purpose		x		
und mehr ... (FPR0-7, MMX0-7, XMM0-7, XMM8-15, FSW, FCW, FTW)							

15.1 Beispiele

HeapAlloc:nBeispiel: x64

```
1# Prolog
2push rbx
3sub rsp,20h
4mov ebx,ecx
5
6# Hauptteil
7test ecx,ecx
8jne factorial+11h (07FF747781011h)
9lea eax,[rcx+1]
10jmp factorial+1Bh (07FF74778101Bh)
11
12# Rekursiver Aufruf
13dec ecx
14call factorial (07FF747781000h)
15imul eax,ebx
16
17# Epilog
18add rsp,20h
19pop rbx
20ret
```

16 Erklärung des Beispiels

- **Prolog**
  - **push rbx**: Sichert den Wert von RBX auf dem Stack.
  - **sub rsp,20h**: Reserviert 32 Byte auf dem Stack.
  - **mov ebx,ecx**: Kopiert den Wert von ECX nach EBX.
- **Hauptteil**
  - **test ecx,ecx**: Überprüft, ob ECX (n) gleich 0 ist.
  - **jne**: Springt zu factorial+11h, wenn n nicht 0 ist.
  - **lea eax,[rcx+1]**: Lädt n+1 in EAX.
  - **jmp**: Springt zu factorial+1Bh.
- **Rekursiver Aufruf**
  - **dec ecx**: Dekrementiert n.
  - **call**: Ruft die Funktion factorial rekursiv auf.
  - **imul eax,ebx**: Multipliziert EAX (Rückgabewert) mit EBX (n).
- **Epilog**
  - **add rsp,20h**: Gibt die reservierten 32 Byte auf dem Stack frei.
  - **pop rbx**: Stellt den ursprünglichen Wert von RBX wieder her.
  - **ret**: Gibt die Kontrolle an den Aufrufer zurück.

17 Calling Convention

Die Verwendung spezifischer Register zur Übergabe von Funktionsargumenten und Rückgabewerten basiert auf der Microsoft x86-64 ABI. Diese Konventionen sind festgelegt, um sicherzustellen, dass Compiler und Linker konsistenten und vorhersehbaren Code erzeugen, der korrekt zwischen verschiedenen Modulen und Bibliotheken funktioniert.

17.1 Registerverwendung für Argumente

- **RCX**: Erstes Argument
- **RDX**: Zweites Argument
- **R8**: Drittes Argument
- **R9**: Viertes Argument

Weitere Argumente werden auf dem Stack abgelegt, beginnend mit dem fünften Argument. Diese Argumente werden in der Reihenfolge von rechts nach links auf den Stack gelegt.

17.2 Rückgabewerte

- Rückgabewerte, die in Register passen (wie Integer oder Zeiger), werden im Register **RAX** zurückgegeben.
- Größere Rückgabewerte, wie Strukturen, werden durch einen vom Aufrufer bereitgestellten Speicherbereich zurückgegeben, wobei ein Zeiger auf diesen Speicherbereich als verstecktes Argument übergeben wird.

17.3 Beispiele

HeapAlloc:

```
1void* HeapAlloc(void* handle, uint32_t flags,
       uint64_t size);
```

- **RCX**: handle
- **RDX**: flags
- **R8**: size

Aufruf der Funktion:

```
1mov rcx, handle    ; handle in RCX
2mov rdx, flags     ; flags in RDX
3mov r8, size       ; size in R8
4call HeapAlloc     ; Funktion aufrufen
```

CreateWindow:

```
1void CreateWindow(const char* className, const char
     * windowName,
2                  uint32_t style, int32_t xpos,
     int32_t ypos,
3                  int32_t width, int32_t height,
     void* parent,
4                  void* menu, void* instance, void*
      userdata);
```

- **RCX**: className
- **RDX**: windowName
- **R8**: style
- **R9**: xpos
- Weitere Argumente (ypos, width, height, parent, menu, instance, userdata) werden auf den Stack gelegt.

Aufruf der Funktion:

```
1mov rcx, className ; className in RCX
2mov rdx, windowName; windowName in RDX
3mov r8, style      ; style in R8
4mov r9, xpos       ; xpos in R9
5sub rsp, 56        ; Platz auf dem Stack für die
     restlichen Parameter
6mov [rsp+48], userdata ; userdata auf dem Stack
7mov [rsp+40], instance ; instance auf dem Stack
8mov [rsp+32], menu     ; menu auf dem Stack
9mov [rsp+24], parent   ; parent auf dem Stack
10mov [rsp+16], height  ; height auf dem Stack
11mov [rsp+8], width    ; width auf dem Stack
12mov [rsp], ypos       ; ypos auf dem Stack
13call CreateWindow ; Funktion aufrufen
```

18 Closures

Definition: Eine Closure ist eine Funktion, die Zugriff auf ihren lexikalischen Gültigkeitsbereich behält, selbst wenn die Funktion außerhalb dieses Bereichs ausgeführt wird. Dies ermöglicht der Funktion, die Umgebung zu "merken", in der sie erstellt wurde.

Theorie:

- **First-Class Functions:** Funktionen können als Rückgabewerte, Parameter oder "Werteeiner Variablen verwendet werden.
- **Funktionsobjekte:** Funktionen als Objekte mit einem Funktionszeiger und Zeiger auf gefangene Variablen in einem Heap-Objekt (Closure-Objekt, CO).
- **Umgebungsproblem:** Bei der Übergabe oder Rückgabe einer Funktion muss die Umgebung die Zerstörung des Stackframes überleben.

Implementierung:

- **Bildung der Closure:** Bei Übergabe oder Speicherung einer Funktion wird eine Closure gebildet, die den Funktionszeiger und gefangene Variablen kapselt.
- **Heap-Objekt:** Gefangene Variablen werden in einem Closure-Objekt auf dem Heap gespeichert, das die Lebensdauer des erzeugenden Stackframes überlebt.
- **Statische Verkettung:** Closure-Objekte haben eine statische Kette, um Zugriffe auf nicht-lokale Variablen zu ermöglichen.

Beispiel in JavaScript:

```
1function outerFunction(outerVariable) {
2    return function innerFunction(innerVariable) {
3        console.log('Outer variable: ' +
         outerVariable);
4        console.log('Inner variable: ' +
         innerVariable);
5    };
6}
7const closureExample = outerFunction('outside');
8closureExample('inside');
```

Beispiel in Python:

```
1def outer_function(outer_variable):
2    def inner_function(inner_variable):
3        print('Outer variable:', outer_variable)
4        print('Inner variable:', inner_variable)
5    return inner_function
6
7closure_example = outer_function('outside')
8closure_example('inside')
```

Beispiel in Rust:

```
1fn outer() -> i32 {
2  let a = 1;
3  let second = || a;
4
5  let first = || {
6    let a = 2;
7    second()
8  };
9
10  first()
11}
12
13fn main() {
14  println!("{:?}", outer());
15}
```

- **Variable Scope:** Die Variable `a` wird in der Funktion `outer` mit dem Wert `1` deklariert.
- **Closure `second`:** Captures die Variable `a` durch Referenz aus dem Gültigkeitsbereich der Funktion `outer`, somit immer `a = 1`.
- **Closure `first`:** Deklariert eine eigene lokale Variable `a` mit dem Wert `2`, ruft aber `second` auf, welches das äußere `a` erfasst.
- **Ausführung:** `first()` ruft `second()` auf, welches `1` zurückgibt.

Common Pitfall: Most Vexing Parse (C++ Beispiel)

```
1void someFunction(float f) {
2  Foo x();
3  x.setVal(f);
4}
5
6int main() {
7  someFunction(2.3);
8  return 0;
9}
10Foo x;
11x.setVal(f);
```

- `Foo x();` wird als Funktionsdeklaration interpretiert, die keine Parameter annimmt und ein `Foo` Objekt zurückgibt.
- Dies führt zu einem Kompilierungsfehler, da `x` kein Objekt ist.
- Lösung: Entfernen Sie die Klammern, um ein Objekt vom Typ `Foo` zu instanziieren.

Upwards Funarg Problem Definition: Das Üpwards Funarg Problem"tritt auf, wenn eine Closure (innere Funktion) zurückgegeben wird, die auf ein Closure-Objekt einer Funktion verweist, die bereits vom Stack gelöscht wurde.

19 Rust: 'fn' vs. 'Fn'

19.1 Funktionszeiger ('fn')

```
1fn add_one(x: i32) -> i32 {
2    x + 1
3}
4
5fn main() {
6    let f: fn(i32) -> i32 = add_one;
7    println!("Result: {}", f(5)); // Ausgabe:
        Result: 6
8}
```

- Repräsentiert eine frei stehende Funktion. - Kann keine Variablen aus dem umgebenden Kontext erfassen.

19.2 Closures ('Fn', 'FnMut', 'FnOnce')

```
1fn call_with_one<F>(f: F) -> i32
2where
3    F: Fn(i32) -> i32,
4{
5    f(1)
6}
7
8fn main() {
9    let closure = |x| x + 2;
10    let result = call_with_one(closure);
11    println!("Result: {}", result); // Ausgabe:
        Result: 3
12}
```

- Repräsentieren anonyme Funktionen, die Variablen aus ihrem umgebenden Kontext erfassen können. - 'Fn': Closure, die keine Variablen verändert. - 'FnMut': Closure, die Variablen verändern kann. - 'FnOnce': Closure, die Variablen konsumiert und nur einmal aufgerufen werden kann.

20 Werte und Typen

20.1 Grundlagen

- **Werte:** Können von einem Programm manipuliert werden.
- **Typen:** Menge von Werten und Operationen auf diesen Werten.
- **Typsystem:** Regeln zur Bindung und Kompatibilität von Typen (statisch, dynamisch).
- Assembler macht nur operationen auf bit gruppen. Es exestieren keine typen.

20.2 Binden von Typen

Statisch:

- Typ wird zur Compilezeit festgelegt.
- Beispiele: C/C++, Java, C#, Haskell.
- ***Typinferenz***: Compiler leitet den Typ aus Initialisierern ab.
  - Herleitung des Typs aus dem Initialisierer durch den Compiler bei der semantischen Analyse
  - Ohne Initialisierung keine Typinferenz!

Dynamisch:

- Typ wird zur Laufzeit implizit durch gebundenen Wert definiert.
- Beispiele: Lisp, Scheme, JavaScript, Perl, Python, Ruby.
- Wert und Typ einer Variable können sich ändern.

	Statische Typen	Dynamische Typen	Statische Typen mit Typinferenz
Typdeklaration	-Umständlich + Klare Kommunikation der Absicht des Entwicklers	+ Unnötig - Erschwert Verständnis der beabsichtigten Verwendung	+ Unnötig o Verwendeter Typ kann leicht oder schwer nachzuvollziehen sein
Typprüfung	-Lückenhaft + Fehlermeldungen zur Compilezeit	- Sehr lückenhaft -Nur in besuchten Zweigen - Fehlermeldung erst zur Laufzeit	-Lückenhaft + Fehlermeldung zur Compilezeit
Laufzeit-Performance	+ Sehr gut	-Performanceverlust durch Typprüfung	+ Sehr gut

20.3 Typprüfung

- **Strongly Typed:** Typfehler werden durch Compiler oder Laufzeitsystem erkannt (z.B. C#, Java, Haskell).
- **Weakly Typed:** Viele Typfehler bleiben unbemerkt (z.B. C, C++), häufig durch zu flexible impliziteKonversionen.
- **Dynamische Typprüfung:** Laufzeit-Typinformation erforderlich (z.B. Python).

20.4 Kompatibilität und Typkonversion

- **Kompatibilität:** Typen müssen oft nur kompatibel sein, nicht identisch.
- **Typkonversion:** Kann explizit (`static_cast<int>(2.0)`) oder implizit (z.B. `int x = 2.0`) erfolgen.
- *Typkompatibilität* wird durch *Typkonversion* erreicht; Explizit, implizit oder benutzerdefiniert

Explizite Konversion	Implizite Konversion
Cast z.B. int x = static_cast<int>(2.0);	Typanpassung = **Coercion** z.B. int x = 2.0;
Converting Cast	**Non-convertig Cast**
Bits werden umgerechnet	Bitmuster bleibt erhalten
Narrowing	**Widening**
Zieltyp kann nicht alle Werte darstellen	Zieltyp ist umfassender
Checked Conversion	**Unchecked Conversion**
Prüfung zur Laufzeit(), ggf. Exception	Keine Prüfung

Die Begriffe Conversion und Cast werden häufig ungenau und oft synonym verwendet

20.5 Primitive Typen

- **Primitive Werte:** Werte, die nicht in einfachere Werte zerlegt werden können (1, true, 'a',...).
  - Wert hängt von Interpretation der Bitgruppe ab
- **Primitive Typen:** Typ dessen Werte primitiv sind
  - Typ ordnet Bitgruppe eine Interpretation zu!
- **Übliche Primitive Typen:** Integer, Character, Boolean, Float.
- **Kardinalität:** Anzahl unterschiedlicher Werte eines Typs (implementierungsabhängig).
- **Benutzerdefinierte Primitive Typen:**
  - **Newtype/Data:** Erzeugt neue Typen basierend auf existierenden Typen (z.B. Haskell, ADA).
  - **Alias:** Neuer Name für existierenden Typ (z.B. `typedef` in C++).

Aufzählungstypen (enum):

- Benutzerdefinierte Typen mit einer endlichen Menge von Werten.
- Werte heißen *enumerands* und können als Zähl- und Indextypen verwendet werden.
- Beispiele:
  - **C/C++:** `enum Days {Mon=1, Tue, Wed, Thu, Fri, Sat, Sun};`
  - **Java:** `enum Day {MONDAY, TUESDAY, ...};`

20.6 Zusammengesetzte Typen

- Zusammengesetzt aus einem oder mehreren primitiven Typen.
- Beispiele: Strukturen, Arrays, algebraische Typen, Objekte, Unions, Strings, Listen, Bäume.

Abbildungen (Mappings)

- Abbildungen bilden Werte aus der Menge S auf die Menge T ab.
- Beispiele: Arrays, mathematische Funktionen.

Arrays

- Indizierte Folge von Komponenten gleichen Typs.
- Zugriff über Index, als Block im Speicher gespeichert.
- Mehrdimensionale Arrays möglich.

Größe des Arrays

- C/C++: Konstante Anzahl Elemente zur Compilezeit.
- Java, C#: Größe bei Allokation festgelegt, danach fest.

Adressrechnung bei Arrays

- **Annahmen:**
  - Index ab 0.
  - 3D rechteckiges Array mit Elementen vom Typ `T`.
  - `a[NZ, NY, NX]` ist ein Array aus:
    - `NZ` Ebenen mit je `NY` Zeilen zu `NX` Elementen.
    - Der letzte Index indiziert hintereinander liegende Elemente ("läuft am schnellsten").
- **Größe in allen Dimensionen:**
  - **Element:** `S1 = sizeof(T)`
  - **Zeile:** `S2 = NX * S1`
  - **Ebene:** `S3 = NY * S2`
- **Adresse von `a[z, y, x]`:**
  - `addressof(a[z, y, x]) = addressof(a) + z * S3 + y * S2 + x * S1`
- **Beispiel:**

```
T a[NZ, NY, NX];
T value = a[z, y, x];
T *p = &(a[z, y, x]);
T value = *p;
T value = *(a + z * S3 + y * S2 + x * S1)
        = *(a + z * NY * NX * sizeof(T)
              + y * NX * sizeof(T)
              + x * sizeof(T))
```

- **Array-Grenzen prüfen:**
  - `0 <= x < NX, 0 <= y < NY, 0 <= z < NZ`

Arraygrenzen Statisch gebunden:

- Compiler speichert Grenzen in der Symboltabelle.
- Code zum Zugriff auf Elemente wird mit diesen Konstanten erzeugt.
- Berechnet effektive Adresse des Elements.
- C/C++ verzichtet auf Prüfung der Arraygrenzen wegen Zeigerarithmetik (Effizienz/Determinismus).
- Rust prüft Arraygrenzen (keine Zeigerarithmetik).

Bei Allokation festgelegt (C#, Java):

- Vor dem Array wird ein Deskriptor angelegt, der für jede Dimension die Größe speichert.
- Code zum Zugriff auf Elemente liest zuerst den Deskriptor.
- Prüfung der Grenzen und Berechnung der effektiven Adresse des Elements.

Collections Generische Collections:

- Arrays sind die historisch wichtigste Collection und sehr effizient.
- Andere Collections: Listen, Bäume, Maps (Dictionaries).
- Zugriff: `[]` ist nur ein Methodenaufruf (z.B. `operator[]` in C++).

std::vector<T> in C++:

- Empfohlener Ersatz für dynamische Arrays.
- Kapselt Array auf dem Heap, dynamisch und sicher (bei Bedarf).
- Speicherverbrauch beachten: Umkopieren wenn reservierter Platz nicht reicht.

```
// Beispiel C++
#include <vector>
using namespace std;

vector<int> a(10); // dynamische Größe
a.push_back(7);    // Append
int v = a[2];      // operator[], kein Check
int w = a.at(2);   // Check
```

Maps:

- Enthalten Schlüssel/Wert-Paare.
- Erlauben oft jeden Typ als Index, meistens Hashtabelle der Schlüssel.
- Oft sind Arrays als Unterform von Maps implementiert (z.B. JS, PHP).

```
// Beispiel JavaScript mit JSON
var x = {
    a: 10,
    b: { z: 100 },
    c: function () { alert('method x.c'); },
    d: [3, 2, 1, "Bummm"]
};
alert(x.a);      // 10
alert(x.b.z);    // 100
x.c();           // 'method x.c'
```

20.7 Wert- und Referenztypen

Werttypen

- Speicherplatz einer Variablen enthält direkt den Wert.
- Wertzuweisung, Parameterübergabe und -rückgabe kopieren den Wert.
- Lebensdauer ist automatisch wie der Stack-Frame.

Referenztypen

- Speicherplatz der Variablen enthält einen Zeiger (Adresse) auf den Wert.
- Wertzuweisung, Parameterübergabe und -rückgabe kopieren nur den Zeiger.
- Lebensdauer des Wertes mindestens solange es Zeiger darauf gibt.

Wann Werttypen?

- **Wertsemantik:** Vergleich nach Wert, keine Objektidentität.
- **Typen:** Primitive Typen (Integer, Float), Strukturen.
- **Feste Größe:** Erlaubt festes Stack-Layout.
- **Kleine Objekte:** Kopieren bei Wertzuweisung erspart Dereferenzieren.

Wann Referenztypen?

- **Objektidentität:** Vergleich von Instanzen, gemeinsame Nutzung.
- **Mutable Objekte:** Arrays, Klassen.
- **Variable Größe:** Dynamische Arrays, Strings.
- **Große Objekte:** Kopieren zu teuer.

Wertsemantik

- **Strings:** Wertsemantik, aber variable Größe.
- **Lösungen:**
  - **Immutable Referenztypen:** Unveränderliche Zeichen.
  - **Immutable Views:** Teile werden ge-shared.
  - **Copy-on-Write (CoW):** Kopie bei Änderung.

Boxing und Unboxing

- **Boxing:** Automatische Konvertierung eines Werttyps in einen Referenztyp.
- **Unboxing:** Rückkonvertierung, um den Wert zu extrahieren.

Escape Analysis

- Ermittelt, ob eine lokale Variable mit Referenztyp nur innerhalb der Prozedur verwendet wird.
- Compiler ersetzt solche Referenztypen durch effizientere Werttypen.

20.8 Zeiger ()

- **Zeiger:** Speichern eine Adresse als Wert.
- **Operatoren:**
  - **Adress-Operator (&):** Liefert die Adresse eines Objekts.
  - **Dereferenzierungs-Operator (*):** Liefert das bezeigerte Objekt.
- **Eigenschaften:**
  - Wertzuweisung, Parameterübergabe und -rückgabe kopieren die Adresse.
  - Typ des Zeigers und des bezeigerten Werts sind verschieden.
  - **L-value** = Left side value (einer Wertzuweisung)
    - Das Objekt selbst, nicht nur sein Wert (als Kopie)
    - Zuweisung an L-value ändert das Objekt

20.9 Heap-Objekte

- **Allokation:** Mit `new` (C++) oder `malloc` (C).
- **Freigabe:** Explizite Speicherfreigabe mit `delete` (C++) oder `free` (C).
- **Syntax:**
  - `->` für Memberzugriff.
  - `(*pr).numerator = 1;` entspricht `pr->numerator = 1;`
- **Probleme:**
  - **Memory Leak:** Lebensdauer > Gültigkeitsbereich, vergessene Freigabe.
  - **Dangling Reference:** Zugriff auf freigegebenen Speicher.

	C++	C#	Java	Python
Struct	Stack	S s;	S s = new S();	keine Structs	keine Structs
Struct	Heap	S *ps = new S();	Structs sind Werttyp (immer)	keine Structs	keine Structs
Klasse	Stack	C c;	Klassen sind Ref.typ	Klassen sind Ref.typ	Klassen sind Ref.typ
Klasse	Heap	C *pc = new C();	C c = new C();	C c = new C();	c = C()

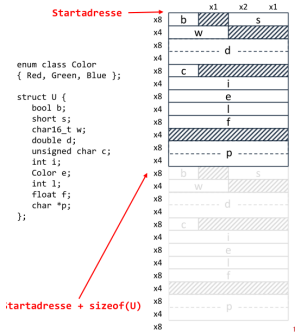
Rust: voll C++ / Heap-allokation nur via Smart-Pointer möglich!
Scriptsprachen: inkl. keine Wahrheitsgröße durch den Programmierer

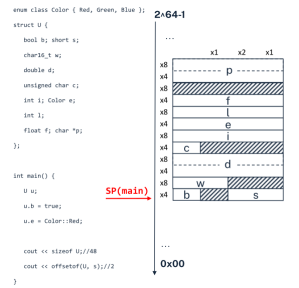
20.10 Zusammengesetzte Typen

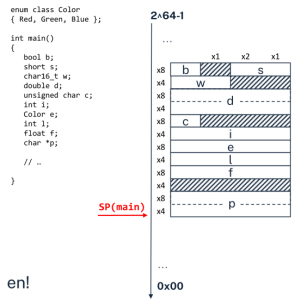
- **Strukturen (Structs):**
  - Gruppierung mehrerer Werte verschiedener Typen.
  - Zugriff über Namen (`struct Point {float x; float y;};`).
- **Tupel:**
  - Gruppierung mehrerer Werte, geordnet nach Position.
  - Kein Zugriff über Namen, nur über Position (`std::tuple<int, float, string>`).
- **Unions:**
  - Speichern verschiedener Typen im gleichen Speicherbereich.
  - Unsicher, da nur ein Wert zur Zeit gültig ist (`union Number {int i; float f;};`).

Speicherlayout

- **Alignment:**
  - **Primitive Typen:** Adresse muss Vielfaches der eigenen Größe (1, 2, 4 oder 8 Byte) sein.
  - **Zusammengesetzte Typen:** Adresse muss Vielfaches des größten primitiven Elements sein.
  - **Padding:** Füllbytes werden eingefügt, um Alignment zu gewährleisten.
- **Heap-Blöcke:**
  - Sind maximal aligned (d.h. 8-fach).
  - Symboltabelle der Struktur speichert Offsets der Elemente.
  - Elemente werden weder gepackt noch umgeordnet.
  - Padding für bündiges Wiederholen (Array of Structs).
- **Stack-Allokation:**
  - Stackframes sind i.d.R. maximal aligned (d.h. 8/16-fach).
  - Symboltabelle der Funktion/Methode speichert Offsets der Variablen.
  - Zugriff auf Elemente lokaler Strukturen: Stack/Frame-Zeiger + Offset der Struktur im Stackframe + Offset des Elements in der Struktur.
- **Beispiel:**
- **Stack-Variablen:**
  - Alignment wie structs.
  - Stackframes sind i.d.R. maximal aligned (d.h. 8-fach).
  - 16-fach Alignment oft bei Prozedureintritt gefordert.
  - Symboltabelle der Funktion/Methode speichert Offsets der Variablen.
  - Kann durch Umordnen optimiert werden.
  - Objekte/Strukturen auf dem Stack müssen in umgekehrter Reihenfolge ihrer Initialisierung zerstört werden.







20.11 Gleichheit von Objekten

- **Bitweise Gleichheit:** Vergleich von Werttypen durch Speicherinhalte.
- **Referenzgleichheit:** Vergleich von Referenztypen durch Zeiger.
- **Referenztypen mit Wertsemantik:** Eigene Implementierung von `operator==` (z.B. in C#).

```
// Beispiel C\#
int i = 5;
int j = 5;
(i == j) // true

int[] a = new int[1];
int[] b = new int[1];
a[0] = 5;
b[0] = 5;
(a == b) // false
(a == a) // true
```

- **Gleichheit rationaler zahlen:** `a/b == c/d` ⇔ `ad == bc`

Heap-Speicherverwaltung und Garbage Collection

- **Manuelle Speicherverwaltung:**
  - `malloc()` in C, `new` in C++.
  - Explizite Freigabe mit `free()`, `delete`.
  - Probleme: Memory Leaks, Dangling References.
- **Reference Counting:**
  - Zählt Zeiger auf ein Objekt.
  - Bei Zähler = 0 wird Objekt freigegeben.
  - Problem: Zyklen.

```
// Beispiel C++
std::shared_ptr<int> p6(new int(5));
std::shared_ptr<int> p7 = p6;
// Beide Zeiger teilen sich das Objekt.
p6.reset(); // Speicher existiert noch, da p7
       zeigt darauf.
p7.reset(); // Speicher wird freigegeben.
```

- **Smart Pointer in C++:**
  - `std::unique_ptr`: Eindeutiger Besitzer.
  - `std::shared_ptr`: Geteilter Besitzer mit Referenzzähler.
  - `std::weak_ptr`: Schwacher Zeiger, erhöht Zähler nicht.

Implementierung

- **Unique-Pointer:**
  - Compiler prüft und ersetzt durch gewöhnliche Zeiger.
- **Shared- und Weak-Pointer:**
  - Handle mit Referenzzählern zwischen Smart Pointer und Objekt.
  - `operator *` überladen für Zugriff auf Heap-Objekt.
  - Konstruktor inkrementiert, Destruktor dekrementiert Zähler.
  - Shared-Zähler auf 0: Objekt freigeben, Weak-Zähler auf 0: Handle freigeben.
  - Lock eines Weak-Pointer gibt Shared-Pointer zurück, falls Shared-Zähler > 0.
- **Optimierung:**
  - Smart Pointer enthält direkten Zeiger auf Heap-Objekt.
  - `make_shared<T>(…)` erzeugt Objekt und Smart-Pointer in einem Schritt.

- **Box<T> in Rust**
  - **Heap-Allokation:** Mit Typ `Box<T>`, ähnlich `std::unique_ptr` in C++.
  - **Eigenschaften:**
    - Kapselt echten Zeiger, besitzt den alloziierten Speicher.
    - Ownership transfer bei Übergabe.
    - Zugriff auf Rohzeiger nur in `unsafe`-Kontexten.
    - Automatische Deallokation bei Verlassen des Gültigkeitsbereichs.

```
// Beispiel in Rust
fn main() {
    let x = Box::new(5);
    // x hat Typ Box<i32>
    // Auf dem Stack -> x (Box<i32>)
    // Auf dem Heap -> 5 (i32)
}
```

- **RC/ARC in Rust**
  - **Reference Counting:** `Rc<T>` für synchrone, `Arc<T>` für asynchrone Implementierungen.
  - **Eigenschaften:**
    - `clone()` erhöht Referenzzähler, macht keine Kopie des Objekts.
    - Beim Verlassen von Gültigkeitsbereichen wird Zähler dekrementiert.
    - Zähler auf 0: Objekt wird aufgeräumt.

```
// Beispiel in Rust
use std::rc::Rc;
struct NamedValue {
    value: i32,
    name: String,
}
fn main() {
    let rc = Rc::new(NamedValue{value: 27,
        name: String::from("approximation")});
    let rc_clone = rc.clone();
    println!("{} = {}", rc.value, rc.name);
}
```

- **Garbage Collection Methoden**
  - **Manuelle Speicherverwaltung:**
    - `malloc()` in C, `new` in C++.
    - Freigabe mit `free()`, `delete`.
  - **Reference Counting:**
    - Zähler für Referenzen auf ein Objekt.

    - Zähler auf 0: Objekt wird freigegeben.
    - Problem: Zyklen.
- **Automatic Reference Counting (ARC):**
    - Compiler generiert Code für Referenzzähler.
    - Weak References können nil werden, wenn Objekt freigegeben.
    - Copy-on-Write bei Referenztypen mit Wertesemantik.

20.12 Dynamische Typen und Laufzeit-Typinformation

Laufzeit-Typinformation

- **Typen und Operationen:** Typen müssen zur Laufzeit erkannt werden, um korrekte Operationen auszuführen. Bitmuster ist string daher konkatenation anstatt additon!
- **Statischer Typ:** Compiler liest Typ aus Symboltabelle und generiert passenden Code.
- **Dynamischer Typ:** RTTI wird im Objekt gespeichert, der Code liest RTTI und passt die Operationen entsprechend an.

Statisch vs. Dynamisch

- **Statisch:** Informationen werden zur Compilezeit in der Symboltabelle gesammelt und in den Code übersetzt.
- **Dynamisch:** Informationen werden zur Laufzeit als Deskriptoren bei den Objekten gespeichert und ausgewertet.

RTTI Implementierung

- **Variable Größe:** Referenztyp, Zeiger auf Heap-Objekt.
- **Optimierungen:** Codierung von bestimmten Werttypen und Zeigern in einem Wort.

Dynamische Typen in compilierten Sprachen

- **Werttypen:** Keine Vererbung, statischer Typ, RTTI unnötig.
- **Polymorphe Typen:** Vererbung, statischer und dynamischer Typ, als Referenztypen oder Zeiger implementiert, RTTI im Objekt.

Typobjekt

- **Typobjekt:** Vom Compiler generiert und als konstante Struktur im Datensegment abgelegt. Enthält Metadaten und Virtual Table.
- **Polymorphe Typen:** Enthalten als RTTI einen Zeiger auf ein Typobjekt.

20.13 Konstanten und Immutable Typen

Arten von "konstant"

- **Compilezeit-Konstanten:** `const` oder literale Konstanten, Speicherplatz fest oder Wert im Code kompiliert.
- **Laufzeit-Konstanten:** `readonly`, `final`, `let`, `val`. Speicherplatz fest nach Konstruktion, keine Mutation.
- **Referenztypen:** Referenziertes Objekt mutable, außer bei Wertsemantik.

Immutable Typen

- **Definition:** Keine Operationen zur Mutation der Objekte. Beispiel: Strings in Java, C#.
- **Implementierung:** Klassen/Strukturen ohne mutierende Member, ggf. neue Objekte erzeugen und zurückgeben.
- **Hinweis:** Variablen immutabler Typen sind i.d.R. nicht konstant.
- **C:** Strings sind mutable Arrays von Zeichen, Zugriff über Index = Zeiger.
- **Java, C#, C++, Swift:** Eigener Typ String, Referenztyp, immutable.
- **Sprachmittel:** In Swift, F#, Scala, aber noch nicht in C#, Java, C++.
- **Vorteile:** Einfach zu konstruieren, zu testen und zu verwenden, thread-safe, keine Copy-Konstruktoren oder Clone-Methoden nötig, Berechnungen cachen, invarieanten bleiben gültig, exceptions hinerlasse keine inkonsistenten Objekte.
- **Nachteil:** Bei Ausdrücken viele neue Objekte, Stress für Speicherverwaltung.

20.14 Generics

Polymorphismus

- **Parametrischer Polymorphismus:** Typ als Parameter, gleiches Verhalten für alle Typen. Typparameter kann explizit, statisch oder dynamisch bestimmt werden.
- **Untertyp-Polymorphismus:** Dynamischer Typ einer Variable kann zur Laufzeit ein Untertyp des statischen Typs sein. Virtuelle Methoden werden dynamisch gebunden.

Generics in C#

- **Konkret:** Typparameter int, string, Student, ...
- **Generisch:** Typparameter T, T1, TKey, ...
- **Datentyp:** `class StudentTable`, `class Dictionary<T1, T2>`
- **Variable:** `StudentTable d;`, `Dictionary<int, Student> d;`
- **Funktion:** `string Stringify(Student st)`
- **Generische Funktion:** `string Stringify<T>(T t)`
- **Delegate:** `delegate string StFunc(Student st)`
- **Anonyme Funktion:** `delegate(Student st) { return st.ToString(); };`

Implementierungen

- **Generischer Zwischencode:** .NET: Zwischencode ist generisch, JIT-Compiler generiert Maschinencode für jeden Typparameter.
- **Type Erasure:** Java: Compiler und JIT generieren Code nur für den Referenztyp `Object`. Alle Werttypen werden geboxt.
- **Code-Ersetzung:** C++: Compiler erzeugt Code für konkreten Typ nur bei Verwendung. Sehr mächtig, aber komplex.

Generics vs. Überladung vs. Überschreiben

- **Generics:** Parametrischer Polymorphismus, 1 Methode, statische Auflösung (Compilezeit).
- **Überladung:** Mehrere Methoden, Auflösung nach Parametern, statische Auflösung (Compilezeit).
- **Überschreiben:** Untertyp-Polymorphismus, dynamische Auflösung (Laufzeit).

21 Rust vs. C Enum

21.1 C Enum

In C sind Enums einfache Aufzählungen von Ganzzahlen:

- Jede Variante entspricht einem Integer-Wert.

- Varianten können keine zusätzlichen Daten enthalten.

21.2 Rust Enum

In Rust sind Enums vielseitiger und können verschiedene Arten von Daten enthalten:

21.2.1 Einfache Varianten

```
enum Direction {
    Up,
    Down,
    Left,
    Right,
}
```

21.2.2 Tupel-ähnliche Varianten

```
enum Event {
    KeyDown(char),      // Enthält ein `char`
    Resize(i32, i32),   // Enthält zwei `i32`
}
```

- Verwenden runde Klammern '()'. - Anonyme Felder. - Gut geeignet für einfache Datentypen ohne spezifische Namen.

21.2.3 Struktur-ähnliche Varianten

```
enum Event {
    Click { x: f32, y: f32 },  // Benannte Felder
    Load,                      // Einfache Variante
}
```

- Verwenden geschweifte Klammern ". - Benannte Felder. - Gut geeignet für komplexere Datenstrukturen, bei denen benannte Felder die Lesbarkeit verbessern.

21.3 Verwendung in Funktionen

Beispiel für die Verwendung eines Enums in einer Funktion:

```
fn handle_event(event: Event) {
    match event {
        Event::KeyDown(c) => println!("Key down event: {}", c),
        Event::Resize(width, height) => println!("Resize event to: {}x{}", width, height),
        Event::Click { x, y } => println!("Click event at coordinates: ({}, {})", x, y),
        Event::Load => println!("Load event"),
    }
}
```

21.3.1 Closure in einem Enum

```
enum Task {
    Simple,
    Complex(Box<dyn Fn(i32) -> i32>), // Box mit einer Closure
}

fn main() {
    let increment = 5;
    let task = Task::Complex(Box::new(move |x| x + increment));

    match task {
        Task::Simple => println!("Simple task"),
        Task::Complex(f) => println!("Result of complex task: {}", f(5)), // Ergebnis: 10
    }
}
```

21.4 Speicherlayout

- Einfache Enums und Enums mit fixen Datentypen werden auf dem Stack gespeichert. - Enums mit dynamischen Daten (z.B. `Box<T>`) speichern diese Daten auf dem Heap, die Enum-Instanz selbst bleibt jedoch auf dem Stack. - Rust reserviert genug Speicher für die größte mögliche Variante plus einen Tag, um die aktuelle Variante zu kennzeichnen.

```
enum Task {
    Simple,
    Complex(Box<dyn Fn(i32) -> i32>), // Box auf dem Heap
}
```

22 Ausdrücke, Anweisungen, Kontrollfluss

22.1 Paradigmen

Turing- vollständig: Sprache mit Wertzuweisung, Sequenz, while, kann alle intuitiv berechenbaren Funktionen berechnen.

- **Sequenz:** `s1; s2;`
- **Sprungbefehle:** `goto`, `return`, ...
- **Bedingte Anweisungen:** `if`, `switch`, ...
- **Schleifen:** `for`, `while`, ...

Komfortabel: Prägnanz und Eleganz der Sprache werden verbessert (aber nicht Mächtigkeit)

- **Prozeduren:** Erlauben prozedurale Abstraktion, Rekursion.
- **Ausnahmen und Behandlung:** Besonderer Kontrollfluss bei Fehlern.
- **Continuations, Coroutinen:** Kontrollflussziele und Umgebung als Daten.
- **Nebenläufigkeit:** Mehrfacher Kontrollfluss zur quasi-parallelen Ausführung.
- **Nichtdeterminismus:** Kontrollflussalternativen ohne Auswahl.

22.2 Sequenz und Sprungbefehle

22.2.1 Sequenzen

- **Funktionale Programmierung:** Keine Anweisungen, nur Ausdrücke. Rückgabewert einer Funktion ist der Wert des letzten Ausdrucks.
- **Imperative / objektorientierte Programmierung:** Programm besteht aus einer Sequenz von Zuweisungen, Ausdrücken, Sprüngen.

22.2.2 Sprünge

- **Goto Considered Harmful:** Alle Sprungbefehle sind bei strukturierter Programmierung entbehrlich.
- **Historisch:** Äquivalent zum JUMP Befehl einer von-Neumann Architektur.
- **Unstrukturierte Sprünge:** Verschleiern Programmstruktur, schlechtere Les- und Wartbarkeit, fehleranfällig.

22.2.3 Pragmatische Sprünge

- Sprünge in andere Gültigkeitsbereiche verboten.
- Sprünge aus aktiven Gültigkeitsbereichen heraus: Lokale Objekte müssen korrekt zerstört werden.
- **Sprungziele:**
    - **break:** Verlassen der innersten Schleife.
    - **continue:** Fortsetzung der nächsten Iteration der innersten Schleife.
    - **return:** Verlassen der Prozedur.
    - **throw:** Ausnahmebehandlung mit ßtack-unwinding".

22.3 Ausdrücke

22.3.1 Ausdrücke/Anweisungen

- **Ausdrücke:** Haben einen Wert (Ergebnis), keine Nebenwirkungen, Auswertungsreihenfolge unwichtig (außer bei Nebenwirkungen, Terminierung, Performance).
- **Anweisungen:** Zweck von Anweisungen sind Nebenwirkungen, Kontrollfluss sehr wichtig.
- **Funktionale Programmierung:** Funktionen und rekursive Funktionen, keine Nebenwirkungen.
- **Imperative Programmierung:** Variablen, Zuweisungen, Prozeduren, Schleifen.

22.3.2 Ausdruck

- **Einfacher Ausdruck:** Name, literale Konstante.
- **Operator mit Operanden:** Symbol als Operator, 0...3 Operanden sind Ausdrücke (Infix-, Präfix-, Postfix-Notation).
- **Funktionsausdruck:** Funktion als Name, Argumente sind Ausdrücke. Präfix-Notation und Klammerung.

22.3.3 Vorrang und Bezug

- **Präzedenz:** Für Operatoren mit verschiedener Priorität.
- **Assoziativität:** Für Operatoren mit gleicher Priorität.
- **Beispiele:** Fortran: Exponentiation hat höchsten Vorrang, ist rechts-assoziativ. C++: `(obj).member` und `obj->member`.

22.4 Zuweisungen

- **Funktionale Sprachen:** Nur Auswertung von Ausdrücken, keine Seiteneffekte.
- **Imperative Sprachen:** Zuweisungen, geordnete Sequenz von Wertänderungen (Seiteneffekt).
- **Seiteneffekt:** Nachfolgende Berechnung wird beeinflusst (unabhängig von Berechnung).
- **Ausdruck vs. Statement:** Ausdruck erzeugt immer einen Wert, Statement existiert nur wegen seines Seiteneffektes.

22.4.1 Seiteneffekte

- **Imperative Sprachen:** "Computing by the means of side effects". Referential Opacity, nicht-deterministisch.
- **Rein funktionale Sprachen:** Keine Seiteneffekte (single assignment). Referential Transparency, deterministisch.

22.4.2 Wertmodell

- **L-Values:** Ausdrücke, die einen Speicherort ergeben. Bilden immer die linke Seite eines Ausdrucks.
- **R-Values:** Ausdrücke, die einen Wert ergeben (das müssen keine Variablen sein). Bilden immer die rechte Seite eines Ausdrucks.
- **Ausdrücke:** Können L-Values oder R-Values sein, abhängig vom Kontext.

22.4.3 Referenzmodell

- **Variable:** Benannte Referenz (Adresse) auf einen Speicherort, anstatt benannter Container für einen Wert. Jede Variable ist damit L-Value.
- **R-Value Kontext:** Muss dereferenziert werden.

22.4.4 Initialisierung

- **Initialisierung:** Erstmaliges Setzen eines Wertes, Aufruf eines Konstruktors (auch bei Kopien).
- **Zuweisung:** Bitweises Kopieren zur Laufzeit, Zuweisungsoperator.
- **Unterscheidung:** Wichtig bei benutzerdefinierten/komplexen Typen. Bei Referenztypen ist der Unterschied nicht relevant (vgl. C#, Java).

22.5 Bedingte Ausdrücke und Anweisungen

22.5.1 Bedingte Ausdrücke

- **Besondere Auswertungsreihenfolge:** Bedingung, dann `then`- oder `else`-Teil.
- **Allgemeine Form:** Ausdruck nach der ersten wahren Bedingung bildet das Ergebnis.
- **Beispiele:**
    - **C-Familie:** `?:`-Operator, `if/case` kein Ausdruck, niedrige Priorität, rechts-assoziativ.
    - **Haskell:** `if`-Ausdrücke, `case-of`-Ausdrücke.
    - **Rust:** `if` als Ausdruck.

22.5.2 Kurzschlussauswertung

- **Reihenfolge:** Linker Operand zuerst, nur falls Ergebnis noch nicht feststeht, rechter Operand.
- **Pseudomaschinencode:** Reihenfolge und Evaluation.

22.5.3 Switch-Anweisung

- **Deklarativ:** 1:n Auswahl aus Alternativen, effizient.
- **Fehleranfällig:** `fallthrough` ohne `break`, fehlendes `default`.
- **Beispiel:**

```
switch(c) {
    case 'A':
        capa++;
    case 'a':
        lettera++;
    default:
        total++;
}
```

22.5.4 Pattern Matching

- **Muster:** Werte, Wertebereiche, Guards, Tupel mit Variablen.
- **Verwendung:** In `if`, `switch`, `while`, `foreach`, etc.
- **Beispiel:**

```
// Rust
struct Point {
    x: i32,
    y: i32,
}
let mut o = Point { x:0, y:2 };
match o {
    Point { x:0, y:0 } => println!("Point is at the origin!"),
    ref mut p @ Point { y:2, .. } => { p.y = 0; println!("y = {}", p.y) },
    Point { x: a, y: b } => println!("{},{}", a, b),
    _ => println!("Got something else!"),
}
```

22.6 Funktionen

22.6.1 Funktionen

- **Begriffe:**
    - **Subroutine/Unterprogramm:** Kontrollfluss kehrt zurück.
    - **Prozedur:** Gibt keinen Wert zurück, Aufruf ist Anweisung.
    - **Funktion:** Gibt einen Wert zurück, Aufruf ist Ausdruck.
    - **Methode:** Gehört zu Klasse oder Objekt.
- **Argumente:** Werden an Unterprogramm übergeben.
- **Parameter:** Variablen im Gültigkeitsbereich des Unterprogramms, an die die Argumente gebunden werden.

22.6.2 Rein funktionale Sprachen

- **Eigenschaften:** Keine Seiteneffekte (Ein-/Ausgabe, Lesen/-Schreiben von nicht-lokalen Variablen).
- **Referential Transparency:** Funktionsaufrufe können ohne das Ergebnis zu verändern durch Funktionswert ersetzt werden.
- **Beispiele:**

```
hello :: [Char] -> [Char]
hello s = s ++ " World!"
main :: IO ()
main = putStrLn \$ hello "Hello"
```

22.6.3 Prozeduren

- **C-Sprachen:** Funktionen == Prozeduren, Prozeduren mit Rückgabe-Typ `void`.
- **OO-Sprachen:** Methoden, freie Funktionen als statische Methoden, teilweise lokale Funktionen/Prozeduren.
- **Skriptsprachen:** Freie Anweisungen werden ausgeführt.

22.6.4 Abstraktionsprinzip

- **Definition:** Jedes zusammengesetzte Konstrukt kann durch eine benannte Abstraktion ersetzt werden.
- **Verwendung:** Abstraktion hat gleiche Semantik wie Konstrukt, wird flexibler durch Parametrisierung.

22.7 Parameterübergabe

22.7.1 Auswertung

- **Reihenfolge:** Operator wird ausgewertet, Operanden/Argumente werden ausgewertet, Operation/Funktion wird ausgeführt.

- **Standard:** Reihenfolge von links nach rechts (C#, Java, JavaScript), in Sprache nicht spezifiziert (C++).
- **Bedingte und boolesche Ausdrücke:** Argumente nach Bedarf ausgewertet, Nebenwirkungen in Argumenten sind schlecht.

22.7.2 Call by Value

- **Definition:** Parameterübergabe wie Wertzuweisung, eine Kopie wird erstellt, der aufgerufene kann das Argument nicht ändern.
- **Referenztyp oder Zeiger:** "call-by-sharing", Mutation des Objekts möglich.
- **Beispiele:**

```
void f1(int a, int b) {
    a = 2;
    b = a + 3;
}
void f2(int a, int b) {
    a = 2;
    b = a + 3;
}
void f3(int a, int &b) {
    a = 4;
    b = a + 3;
}
int main() {
    int x = 1;
    int y = 2;
    f1(x, y);  // x == 1, y == 2
    f2(&x, y); // x == 2, y == 2
    f3(x, y);  // x == 2, y == 7
    return 0;
}

```

22.7.3 Call by Reference

- **Definition:** Zeiger auf Argument wird erzeugt und übergeben, implizite Dereferenzierung im Rumpf.
- **Argument:** L-Value, sinnvoll, da Aufrufer die Änderung verwenden kann.
- **R-Value References in C++:** Objekt kann ausgebeutet"werden, da Aufrufer keinen Zugriff mehr hat (Effizienz).
- **Beispiele:**

```
void incr(int &x) {
    x = x + 1;
}
int main() {
    int i = 1;
    incr(i);  // i == 2
}

```

22.7.4 Vergleich

- **Call-by-Sharing vs. Call-by-Reference:** Referenzierte Objekte können nur mutiert, aber nicht ersetzt werden.
- **Beispiel:**

```
static void Swap<T>(ref T x, ref T y) {
    T tmp = x;
    x = y;
    y = tmp;
}
int i = 3;
int j = 5;
Swap(ref i, ref j);
// i == 5, j == 3

```

22.7.5 Schutz vor Mutation

- **Problem:** Wie übergibt man große Argumente (z.B. Arrays), die nicht geändert werden dürfen?
- **Lösungen:**
  - **Call-by-Value und Arrays mit Wertsemantik:** `std::vector` in C++, Copy on Write.
  - **Call-by-Reference und `const`:** `const Object&` in C++, Immutable Typen, `const`-Wrapper Klassen.
- **Hinweis:** `const` ist "Default"bei neueren Sprachen, Möglichkeit zur Mutation muss explizit angegeben werden.

22.7.6 Call-by-Name

- **Definition:** Argumente werden erst bei Bedarf im Rumpf ausgewertet, statisches Binden freier Variablen.
- **Implementierung:** Durch Einpacken in implizite Closures, Nachbilden in anderen Sprachen durch explizite Closures.
- **Beispiel:**

```
// Haskell
myif :: Bool -> a -> a -> a
myif True x _ = x
myif False _ y = y
myif True 5 (1/0) // -> 5

```

22.7.7 Trailing Closures

- **Definition:** Block hinter Funktionsaufruf wird als "Closure"übergeben.
- **Beispiel:**

```
// Swift
func dumpTime(message: String, action: () ->
        ()) {
    let start = NSDate()
    action()
    let end = NSDate()
    println(message + "\(end.
       timeIntervalSinceDate(start))")
}
var a: [Int] = []
dumpTime("fillArray") {
    for i in 0..<1000000 { a.append(i) }
}

```

22.8 Schleifen

22.8.1 Schleife vs. Rekursion

- **Schleife:** Iteration bis boolesche Bedingung zutrifft.
- **Rekursion:** Alternative zur Schleife, nützlich für Akkumulation.
- **Varianten:**
  - Test am Anfang der Schleife.
  - Test am Ende der Schleife.
  - 1½ Schleife.
  - C-Style-for ist mächtig wie while, kann Endlosschleifen.

22.8.2 Iteratoren

- Iteration über alle Elemente einer Collection.
- Jede Collection sollte Iterator(en) zur Verfügung stellen.
- Verschiedene Iteratoren: vorwärts, rückwärts, pre-/in-/post-order, alphabetisch, usw.
- `foreach`-Schleife verbirgt Iterator.
- Rust, Python: keine generische `for (;;)` Schleife mehr.

22.8.3 Map / Reduce

- Funktionale Iteration: Map (Projektion) und Reduce (Akkumulation).
- Innere Iteration: Iteration wird in der Collection verborgen.
- Funktionaler Stil: Funktionen definieren Ergebnis, nicht Ablauf.

22.8.4 Generatoren

- Implementierung von Iteratoren mit `yield`.
- `yield` liefert Element der Iteration, Kontext (Wert von i, Programmzähler) bleibt erhalten.

22.8.5 Implementierung

- Iterator ist innere Klasse der Collection.
- Generator wird in innere Klasse übersetzt.
- Schleifen werden zu `goto`-Programm.
- Programm wird mit `switch-case` an den `yield`-Stellen segmentiert.
- Lokale Variablen werden zu Instanzvariablen, Programmzähler wird zu _state.

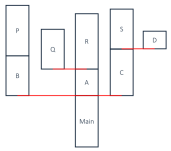
22.9 Coroutinen

22.9.1 Yield Coroutinen

- Generalisierung von Funktionen, erlauben expliziten Austritt und Wiedereintritt.
- Zustände: Erzeugt, Suspendiert, Aktiviert, Zerstört.
- `yield` gibt Kontrollfluss ab, Stackframe/Umgebung wird aufbewahrt.

22.9.2 Coroutinen

- **Asymmetrische Coroutine:** `yield` übergibt Kontrollfluss zwischen beliebigen Coroutinen, Fortsetzung mit `resume`.
- **Symmetrische Coroutine:** Coroutine ist Aufrufer untergeordnet, Kontrollfluss nur zurück an Aufrufer.
- **Nebenläufigkeit:** Gleichzeitig gestartet, aber nicht beendet bzw. laufend. Nur eine Coroutine zu einem bestimmten Zeitpunkt aktiv.
- **Stackful Coroutine:** `yield` kann in beliebigem Unterprogramm vorkommen, Stack wird aufbewahrt.
- **Stackless Coroutine:** `yield` nur in Coroutine selbst, Implementierung durch Transformation in ein Objekt mit Zustand.



- Main ruft A auf und erzeugt COs B und C
- B ruft P auf und erzeugt CO Q
- C ruft S auf u nd erzeugt CO d

22.9.3 Promise / Future

- Ereignis löst Interrupt aus, ruft globalen Handler auf.
- Moderne Implementierung kapselt globale Handler: Objekte bzw. instanzlokale Callbacks.
- **Problem:** "Callback Hell"bei geschachtelten Callbacks.
- **Lösung:** Kapsle asynchrone Operation in Objekt, Interface erhält Callback für Operation, Instanz stellt Methoden für Ergebnis- und Fehlerbehandlung bereit.

22.9.4 Async / Await

- Syntactic Sugar für Promises (JavaScript), Coroutinen (C++), Task API (C#), `std::future::Future` (Rust).
- Ersetzt `yield` in Python komplett.
- Man kann asynchronen Code wie sequenziellen schreiben, da man ohnehin wartet, erübrigen sich Synchronisationspunkte.

22.10 Rekursion

- **Mächtigkeit:** Genauso mächtig wie Iteration.
- **Intuitivität:** Oft intuitiver als iterative Variante.
- **Effizienz:** Naive Implementierung oft weniger effizient.

22.10.1 Endrekursion (Tail Recursion)

- **Definition:** Rekursiver Aufruf als letzter Berechnungsschritt.
- **Optimierung:** Tail Recursion Elimination durch Rücksprung ohne neuen Stackframe.
- **Beispiel:**

```
int gcd(int a, int b) {
    start:
    if (a == b)
        return a;
    else if (a > b)
        return gcd(a-b, b);
    else
        return gcd(a, b-a);
}

```

Tail Call Optimization Verallgemeinerung der Tail Recursion Elimination

- **Definition:** Prozeduraufruf als letzter Berechnungsschritt.
- **Vorgehensweise:**
  - Präparieren der Parameter auf dem Stack für das Sprungziel.
  - Unterprogrammsprung (`call`) durch `goto` (`jmp`) ersetzt.
  - Rücksprung (`return from subroutine`) geht direkt zum gemeinsamen Aufrufer.
  - Sprung im Dreieck".
- **Beispiel:**
- **Vorteil:** Spart neuen Stackframe, effizienter Rücksprung.
- **Beispiele:** A und C (im Beispiel) sind unverändert.

22.10.2 Tail Call Optimization

- **Definition:** Prozeduraufruf als letzter Berechnungsschritt.
- **Implementierung:** Unterprogrammsprung (call) durch `goto` (jmp) ersetzt.
- **Vorteil:** Rücksprung geht direkt zum gemeinsamen Aufrufer.
- **Beispiel:**

22.10.3 Optimierung: Inlining

- **Definition:** Präparieren der Parameter als neue lokale Variablen, Einkopieren des Rumpfes der Prozedur statt dem Aufruf.
- **Vorteil:** Spart Unterprogrammsprung.
- **Nachteil:** Bläht Code auf.
- **Beispiel:**

22.11 Exceptions

22.11.1 Fehlerstrategien (Traditionell)

- **Externe Bibliotheken:** Eigene Fehlerstrategie.
- **Reservierte Rückgabewerte:** `1`, `"null"`, Status als Rückgabewert.
- **Probleme:**
  - Status kann ignoriert werden.
  - Kann sich mit gültigen Rückgabewerten überschneiden.
  - Keine Nutzung in Ausdrücken möglich.
  - Integration verschiedener Fehlerstrategien erforderlich.

22.11.2 Exception Handling

- **Robuste Anwendungen:** Arbeiten unter vielen Fehlersituationen weiter. Flugzeug computer sollte bei fehler nicht abstürzen.
- **Nicht regulärer Programmablauf:** Falsche Benutzereingaben sind keine Exception.
- **Beispiele:**
  - PageFault oder automatische Garbage Collection sind keine Ausnahmen.
  - Out of memory schon.
- **Trennung:** Regulärer und Fehlerpfad im Programmtext.

22.11.3 Catch-Blöcke

- **Ausführung:** Anhand des Typs des geworfenen Objekts wird der erste passende in der Aufrufhierarchie ausgeführt.
- **Konsumierung:** Exception ist konsumiert (außer weiteres `throw;` im Handler).
- **Finally-Block:** Wird immer ausgeführt (C#, Java).
- **Filter-Block (MSIL/CIL):** Prüft Exception ohne Konsumieren.
- **Fault-Block (MSIL/CIL):** Jede Exception passiert den Fault-Block.
- **Beispiel:**

```
static void Main(string[] args) {
    try {
        Console.WriteLine("Kehrwert ist {0}",
       StringMath.Reziprocal(args[0]));
        Console.WriteLine("Produkt ist {0}",
       StringMath.Multiply(args[0], args[1]));
    }
    catch (FormatException) {
        Console.WriteLine("You did not enter a
       number.");
    }
    catch (ArgumentNullException) {
        Console.WriteLine("You did not supply
       any input.");
    }
    catch (OverflowException) {
        Console.WriteLine("The value entered
       is out of range.");
    }
    catch (NotImplementedException) {
        Console.WriteLine("Functions is not
       yet implemented.");
    }
    catch (Exception e) { // catch rest
        Console.WriteLine("Message = {0}", e.
       Message);
        Console.WriteLine("Source = {0}", e.
       Source);
        Console.WriteLine("StackTrace = {0}",
       e.StackTrace);
        Console.WriteLine("TargetSite = {0}",
       e.TargetSite);
    }
    finally {
        Console.WriteLine("Good Bye!");
    }
}

```

22.11.4 Typen in Exceptions (Beispiele)

- **C++:** Alle Typen können geworfen werden.
- **C#:** Alle, die von `System.Exception` erben.
- **Java:** Alle, die von `Throwable` erben. Checked vs. unchecked exceptions.
- **Python:** Alle, die von `Exception` erben. `raise` statt `throw`.
- **Haskell:** Imprecise vs. Precise Exceptions. Handling nur im IO Monad.
- **Rust:** Hat keine Exceptions. Alg. Datentypen und `panic!`.

22.11.5 Deklaration von Exceptions

- **Formale Schnittstelle:** Exceptions gehören zur Schnittstelle.
- **Java:** "Checked exceptions"müssen deklariert werden. Exception-Poisoning bei nicht behandelten.
- **C++:** `throw()` Spezifizierer "überholtßeit C++11, entfernt in C++17/20. Stattdessen: `noexcept` Spezifizierer/Operator.
- **C# und andere:** Keine Deklaration nötig.

22.11.6 Implementierung

- **Tabellen:** Compiler/Linker erzeugen Tabellen im Datensegment (konstant).
- **Zuordnung:** Programmzählerbereiche werden den Handlern zugeordnet.
- **Stack Unwinding:** Rücksprungadressen werden gelesen, um die Programmzähler des Aufrufers zu finden.
- **Kosten:** Geringe Kosten zur Laufzeit (z.B. wg. Codegröße der Handler), solange keine Exceptions auftreten.
- **Größere Executables:** Codegröße der Handler erhöht Executable-Größe.
- **Wahl:** In C++ wählbar.
- **Beispiel:**

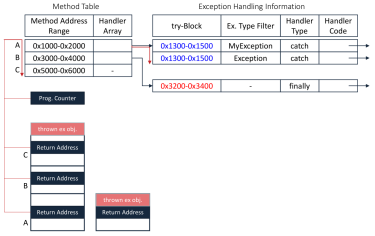
```
class MyException : Exception { }

void A() {
    try {
        B();
    } catch (MyException e) {
        Console.WriteLine(e);
    } catch (Exception e) {
        Console.WriteLine(e);
    }
}

void B() {
    FileStream f = new FileStream("myfile.txt"
       , FileMode.Create);
    try {
        C();
    } finally {
        f?.Close();
    }

static void C() {
    throw new Exception();
}

```

Beispiel (.NET)

- **Ablauf:**
  - Suche nach erstem Programmzähler/Return Address, die Methode mit passendem Catch-Handler hat.
  - `try`-Blöcke sind von innen nach außen sortiert.
  - Rufe alle `finally`-Handler von Programmzähler bis zum Catch-Handler auf.
  - Der Framepointer (nicht der Stack Pointer) wird passend herabgesetzt.
  - Lösche Stackframes bis zum Aufruf des Catch-Handlers.
  - Das Exception-Objekt wird beim Abstieg kopiert.
- **Method Table:**

```
Method Address Range  Handler Array
20x1000-0x2000        MyException catch
30x3000-0x4000        Exception catch
40x5000-0x6000        - finally
5
```

- **Visualisierung:**



23 Objektorientierte Programmierung

23.1 Konzepte

23.1.1 Kapselung

- **Modul:** Gruppe von Konstanten, Variablen, Prozeduren und Typen mit expliziter Schnittstelle (import/export=public).
- **Komponente:** Konstanten wie Strings, Bilder, etc. werden zu Ressourcen aufgewertet.
- **Instanziierung:** Module sind selbst Typen, dadurch mehrere Instanzen möglich.

23.1.2 Vererbung

- **Hierarchie:** Anordnen der Typen in einer Hierarchie.
- **Untertyp:** Spezialisierung des Obertyps.
- **Substitutionsprinzip:** Instanz eines Untertyps kann Instanz des Obertyps ersetzen.

23.1.3 Dynamisch und Statisch typisierte Sprachen

- **Dynamisch:** Variable hat keinen Typ, laufzeitabhängiger Wert hat Typ.
- **Statisch:** Variable hat statischen Typ, laufzeitabhängiger Wert hat gleichen Typ oder Untertyp.

23.1.4 Dynamisches Binden von Methoden

- **Dynamisch typisierte Sprachen:** Late Binding, Lookup in Methodentabelle des Objekts.
- **Statisch typisierte Sprachen:** Dynamic Dispatch, Sprung über den n-ten Eintrag der vtable des Objekts.

23.1.5 Statisches vs. dynamisches Binden von Methoden

- **Statisches Binden:** Methode wird durch den statischen Typ der Variable festgelegt.
- **Dynamisches Binden:** Methode wird durch den dynamischen Typ der Variable festgelegt.

23.2 Dynamische Typen

- **Variablen:** Sind typlos, Typ wird bei Konstruktion des Objekts angegeben.
- **Operatoren:** Eingebaute Operatoren lösen zur Laufzeit ggf. Typfehler aus.

23.2.1 Late Binding

Dynamisch typisierte Sprachen

- **Compilezeit-Checks:** Keine Checks, ob Methode existiert.
- **Duck-Typing:** Nur Phänotyp ist wichtig, kein Genotyp.
  - `"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."`
  - JavaScript, Smalltalk, Python, Objective-C, C# dynamic, PHP, Ruby, u.v.m. ...

23.3 Untertyp-Polymorphismus

23.3.1 Dynamischer Typ

- **Laufzeitsystem und GC:** Referenztypen enthalten Laufzeit-Typinformation (RTTI).
- **Typsicher:** Stets Run-time Check des Typs beim Downcast.
- **C++:** Nur Structs/Klassen mit virtuellen Methoden enthalten RTTI.

23.3.2 C++ Konversionen

- **dynamic_cast:** Prüft dynamischen Typ, ist sicher.
- **static_cast:** Prüft nur manche Werte des statischen Typs, unsicher.
- **reinterpret_cast:** Kopiert die Bits, unsicher.
- **C-Style Cast:** Wie static_cast, Fallback auf reinterpret_cast, unsicher.

23.4 (Virtuelle) Methoden

23.4.1 Klassenmethoden

- **Im Namensraum der Klasse:** Klassenmethode ist Funktion.
- **Instanzmethode:** Funktion mit extra this-Parameter, Referenz/Zeiger auf Instanz als impliziter 1. Parameter.

23.4.2 Virtuelle Methoden

- **Wahlbar:** Performance schlechter.
- **Logisch:** Bei Untertyp-Polymorphismus sind statisch gebundene Methoden falsch.

23.4.3 Implementierung

- **Instanzmethode:** Wird durch Indizierung der vtable ausgewählt.

```cpp
// C++
class MyClass {
    int i;
    virtual void foo() { i = i + 1; }
};

MyClass obj = new MyClass();
obj.foo();

// Func ist universeller F.ptr.
struct MyClass {
    Func[] vtable;
    int i;
};

void foo(ref MyClass this) {
    this.i = this.i + 1;
}

Func[] myClass_vtable = { foo };
const int foo_id = 0;
void MyClass_ctor(ref MyClass this) {
    this.vtable = myClass_vtable;
    this.i = 0;
}

MyClass obj = new MyClass();
MyClass_ctor(ref obj);
obj.vtable[foo_id](ref obj);
```

24 C# Closures und Predicate

Definition:

- `Predicate<T>`: Ein vordefinierter Delegattyp, der eine Methode darstellt, die ein Argument vom Typ `T` akzeptiert und einen booleschen Wert zurückgibt.

Funktion `And`:

```cpp
static Predicate<T> And<T>(Predicate<T> p1,
        Predicate<T> p2) {
  return (T t) => p1(t) && p2(t);
}
```

- Kombiniert zwei `Predicate<T>` mit dem logischen UND-Operator.

Implementierung von `MakeIntervalTest`:

```cpp
static Predicate<double> MakeIntervalTest(double
        min, double max) {
  Predicate<double> notLessMin = (double x) => min
        <= x;
  Predicate<double> notGreaterMax = (double x) => x
        <= max;
  return And(notLessMin, notGreaterMax);
}
```

- Verwendet `And`, um `min <= x <= max` zu prüfen.

Closures im Programm:

- Erzeugt an drei Stellen:
  - `(double x) => min <= x` fängt `min`.
  - `(double x) => x <= max` fängt `max`.
  - `(T t) => p1(t) && p2(t)` fängt `notLessMin` und `notGreaterMax`.

Anzahl der gebildeten Closures: 3 **Anzahl der gefangenen Variablen oder Parameter:** 4

C++

Programmcode:

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

int main() {
  const int N = 2;

  std::vector<double> v(N);
  int i = 1;

  std::transform(v.begin(), v.end(), v.begin(),
    [&i](double e) {
      return 1.0 / i++;
    }
  );

  double s = std::accumulate(v.begin(), v.end(),
       0.0,
    [] (double e1 , double e2) {
      return e1  +  e2 ;
    }
  );

  std::cout <<  s  <<  std::endl;

  return 0;
}
```

Analyse:

- Ausgabe des Programms: `1.5`
- `std::transform`: Transformiert Elemente des Vektors `v` mit dem Lambda `[&i](double e) { return 1.0 / i++; }`.
- `std::accumulate`: Summiert die Elemente des Vektors `v` mit dem Lambda `[](double e1, double e2) { return e1 + e2; }`.
- Closures werden an zwei Stellen gebildet:
  - Lambda in `std::transform` fängt `i` ein.
  - Lambda in `std::accumulate` fängt keine Variablen ein.

Implementierung ohne Lambda-Ausdrücke:

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

class Transform {
public:
  Transform(int\& i) : i(i) {}
  double operator()(double e) {
    return 1.0 / i++;
  }
private:
  int\& i;
};

class Accumulate {
public:
  double operator()(double e1, double e2) const {
    return e1 + e2;
  }
};

int main() {
  const int N = 2;

  std::vector<double> v(N);
  int i = 1;

  std::transform(v.begin(), v.end(), v.begin(),
       Transform(i));

  double s = std::accumulate(v.begin(), v.end(),
       0.0, Accumulate());

  std::cout << s << std::endl;

  return 0;
}
```

24.1 Exceptions

Regel E.15: "Catch exceptions from a hierarchy by reference"

- **Polymorphismus:** Fangen durch Referenz bewahrt die tatsächliche Typinformation der Ausnahmeobjekte.
- **Vermeidung von Kopien:** Effizienter und vermeidet potenzielle Probleme.
- **Vererbungshierarchie:** Verhindert den Verlust spezifischer Informationen abgeleiteter Ausnahmen.

Beispiel:

```cpp
class BaseException : public std::exception {};
class DerivedException : public BaseException {};

try {
  // Code, der eine DerivedException wirft
} catch (const BaseException& e) {
  // catches auch alle abgeleiteten Klassen von
       BaseException
}
```

Beispiel:

```cpp
void f() {
  try {
    // ...
  }
  catch (Derived& d) { /* ... */ }  // Spezieller
  catch (Base& b) { /* ... */ }     // Allgemeiner
  catch (std::exception& e) { /* ... */ } //
       Standardausnahmen
  catch (...) { /* ... */ }          // Alle
       anderen Ausnahmen
}
```

24.2 Dekriminator und vergleich

Code:

```cpp
#include <iostream>

int main() {
  int x = 10;
  while (x --> 0) {
    std::cout << x << std::endl;
  }

  return 0;
}
```

Erklärung des Operators `x --> 0`:

- Der Ausdruck `x --> 0` ist eine Kombination aus dem Postfix-Dekrementoperator `x--` und dem Vergleichsoperator `>`.
- `x--` verringert den Wert von `x` um 1, gibt aber zuerst den aktuellen Wert von `x` zurück.
- Der Vergleich `>` prüft dann, ob der ursprüngliche Wert von `x` größer als 0 ist.

Ablauf der Schleife:

- `x = 10`, `x-- > 0` wird als `10 > 0` ausgewertet, `x` wird auf 9 verringert.
- `9` wird ausgegeben.
- Dieser Vorgang wiederholt sich, bis `x` auf 0 verringert wird.
- Wenn `x = 0`, `x-- > 0` wird als `0 > 0` ausgewertet, die Schleife endet.

25 Spaceship-Operator (<=>)

Zweck:

- Vereinfachung der Implementierung von Vergleichsoperationen.
- Definiert alle sechs Vergleichsoperatoren (`<`, `<=`, `>`, `>=`, `==`, `!=`) gleichzeitig.
- Reduziert Fehler und erhöht die Konsistenz.
- Automatische Generierung der anderen Vergleichsoperatoren durch den Compiler.
- Unterstützt starke, schwache und teilweise Ordnung.

Beispiel:

```cpp
#include <iostream>
#include <compare>

class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}
    auto operator<=>(const Point& other) const =
        default;
};

int main() {
    Point p1(1, 2);
    Point p2(2, 3);
    if (p1 < p2) {
        std::cout << "p1 is less than p2\n";
    } else if (p1 > p2) {
        std::cout << "p1 is greater than p2\n";
    } else {
        std::cout << "p1 is equal to p2\n";
    }
    return 0;
}
```

26 Iterators

26.1 Python

```python
class Range:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __iter__(self):
        return self
    def __next__(self):
        if self.a >= self.b:
            raise StopIteration
        if self.a % 2 != 0:
            self.a += 1
        current = self.a
        self.a += 2
        return current

range = Range(1, 21)
for num in range:
    print(num)
```

26.2 JavaScript

```javascript
function makeUnevenIterator(a, b) {
  let nextValue = a;
  const unevenIterator = {
    next() {
      while (nextValue <= b) {
        if (nextValue % 2 != 0) {
          let rValue = nextValue;
          nextValue += 1;
          return { value: rValue, done: false };
        }
        nextValue += 1;
      }
      return { done: true };
    },
    [Symbol.iterator]() { return this; }
  };
  return unevenIterator;
}

let it = makeUnevenIterator(0, 20);
for (let v of it) {
  console.log(v);
}
```

26.3 Rust

```rust
struct Range {
    a: i32,
    b: i32,
}

impl Range {
    fn new(a: i32, b: i32) -> Range {Range { a, b
        }}
}

impl Iterator for Range {
    type Item = i32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.a > self.b {
            return None;
        }
        if self.a % 2 != 0 {
```

```
            self.a += 1;
        }
        if self.a <= self.b {
            let curr = self.a;
            self.a += 2;
            Some(curr)
        } else {
            None
        }
    }
}

fn main() {
    let mut counter = Range::new(1, 21);
    while let Some(num) = counter.next() {
        println!("{}", num);
    }
}
```

26.4 Generator

27 Endrekursion

- Eine Funktion ist **endrekursiv** (tail recursive), wenn der rekursive Aufruf die letzte Operation in der Funktion ist.
- Vorteil: Endrekursive Funktionen können von Compilern optimiert werden, um Speicherplatz und Rechenzeit zu sparen, indem sie rekursive Aufrufe in Schleifen umwandeln.

Einfache rekursive Implementierung der Fibonacci-Funktion:

```
-- Einfache rekursive Fibonacci-Funktion
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

main :: IO ()
main = do
    putStrLn "Geben Sie eine Zahl ein:"
    input <- getLine
    let n = read input :: Int
    print (fib n)
```

Endrekursive Implementierung der Fibonacci-Funktion:

```
-- Hilfsfunktion fuer endrekursive Fibonacci-Berechnung
fib' :: Int -> Int -> Int -> Int
fib' 0 a _ = a
fib' n a b = fib' (n - 1) b (a + b)

-- Endrekursive Fibonacci-Funktion
fib :: Int -> Int
fib n = fib' n 0 1

main :: IO ()
main = do
    putStrLn "Geben Sie eine Zahl ein:"
    input <- getLine
    let n = read input :: Int
    print (fib n)
```

28 Promise vs. Future in C++ und Rust

Definitionen:

- **Promise:** Ein Objekt, das einen Wert oder eine Ausnahme repräsentiert, die möglicherweise zu einem späteren Zeitpunkt bereitgestellt werden.
- **Future:** Ein Objekt, das einen Wert oder eine Ausnahme repräsentiert, die möglicherweise in der Zukunft bereitgestellt werden.

Typen:

- C++: `std::promise<T>`, `std::future<T>`
- Rust: `std::future::Future`

C++ Beispiel:

```
#include <iostream>
#include <thread>
#include <future>

void printPromiseValue(std::future<int>& fut) {
    std::cout << "Wert vom Promise: " << fut.get()
       << std::endl;
}

int main() {
    std::promise<int> prom;
    std::future<int> fut = prom.get_future();

    std::thread t(printPromiseValue, std::ref(fut))
       ;

    std::this_thread::sleep_for(std::chrono::
       seconds(1));
    prom.set_value(42);

    t.join();

    return 0;
}
```

Rust Beispiel:

```
use std::thread;
use std::time::Duration;
use futures::executor::block_on;

async fn compute_value() -> i32 {
    thread::sleep(Duration::from_secs(1));
    42
}

async fn print_future_value() {
    let value = compute_value().await;
    println!("Wert vom Future: {}", value);
}

fn main() {
    block_on(print_future_value());
}
```

JavaScript Vergleich:

```
function computeValue() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(42);
    }, 1000);
  });
}

computeValue().then((value) => {
  console.log("Wert vom Promise: " + value);
});
```

Erklärung:

- **C++:** Ein `std::promise<int>` erstellt ein `std::future<int>`. Ein separater Thread wartet auf das Future und gibt den Wert aus, wenn er gesetzt wird.
- **Rust:** Eine asynchrone Funktion wird definiert und aufgerufen. Der Executor blockiert, bis das Ergebnis verfügbar ist.
- **JavaScript:** Ein Promise wird erstellt, das nach einer Sekunde aufgelöst wird. Das `then`-Handler gibt den Wert aus.

29 Altklausuren

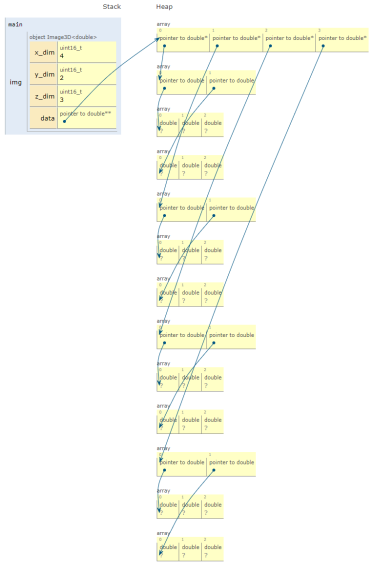
29.1 Aufgabe 1

```
        #include <cstdint>

        template <typename T>
        class Image3D {
        public:
            Image3D(uint16_t xd, uint16_t yd,
        uint16_t zd);

            /* ... Weitere Deklarationen ...
        */

        private:
            T*** data;

            /* ... Weitere Deklarationen ...
        */
        };
```

1. **Was bedeutet `template <typename T>` in Zeile 3 der Deklaration?**
   `template <typename T>` in Zeile 3 bedeutet, dass die Klasse `Image3D` ein Template ist und einen Typ-Parameter `T` akzeptiert. Dies erlaubt die Erstellung von `Image3D`-Objekten mit unterschiedlichen Datentypen.
2. **Welchen Typ hat die Membervariable `data`?**
   Die Membervariable `data` ist ein dreidimensionales Array von Zeigern auf `T`. In Worten: `data` ist ein Zeiger auf einen Zeiger auf einen Zeiger auf Typ `T`.
3. **Geben Sie eine mögliche Implementierung des Konstruktors an.**

```
#include <cstdint>
#include <iostream>
template <typename T>
class Image3D {
public:
    Image3D(uint16_t xd, uint16_t yd, uint16_t
       zd) : x_dim(xd), y_dim(yd), z_dim(zd)
       {
        data = new T**[x_dim];
        for (uint16_t i = 0; i < x_dim; ++i) {
            data[i] = new T*[y_dim];
            for (uint16_t j = 0; j < y_dim; ++
       j) {
                data[i][j] = new T[z_dim];
            }
        }
    }

    ~Image3D() {
        for (uint16_t i = 0; i < x_dim; ++i) {
            for (uint16_t j = 0; j < y_dim; ++
       j) {
                delete[] data[i][j];
            }
            delete[] data[i];
        }
        delete[] data;
    }

private:
    uint16_t x_dim, y_dim, z_dim;
    T*** data;
};
```

4. **Benötigt die Klasse `Image3D` einen benutzerdefinierten Destruktor? Begründen Sie Ihre Antwort.**
   Ja, die Klasse benötigt einen benutzerdefinierten Destruktor, um den dynamisch alloziierten Speicher freizugeben und Speicherlecks zu vermeiden.
5. **Wieviele Allokationen müssen durchgeführt werden, um die Instanz `img` zu erzeugen?**
   Um die Instanz `img` zu erzeugen, sind 9 Allokationen notwendig (1 für das `data` Array, 4 für die `T**` Arrays, 4 für die `T*` Arrays).
6. **Zeichnen Sie ein Abbild des Stacks und des Heaps, das sich unter Berücksichtigung Ihrer Implementierung ergibt.**



7. **Wieviele Allokationen benötigt man mindestens für eine äquivalente Implementierung?**
   Man benötigt mindestens eine Allokation, um den gesamten Speicherplatz auf einmal zu reservieren.
8. **Welche Änderungen muss man vornehmen, um die Anzahl der Allokationen zu reduzieren?**
   Man könnte ein zusammenhängendes Speicherblock-Array verwenden. Änderungen im Code könnten wie folgt aussehen:

```
#include <cstdint>
#include <iostream>
template <typename T>
class Image3D {
public:
    Image3D(uint16_t xd, uint16_t yd, uint16_t
       zd) : x_dim(xd), y_dim(yd), z_dim(zd)
       {
        data = new T[x_dim * y_dim * z_dim];
    }

    ~Image3D() {
        delete[] data;
    }

    T& at(uint16_t x, uint16_t y, uint16_t z)
       {
        return data[(x * y_dim * z_dim) + (y *
       z_dim) + z];
    }

private:
    uint16_t x_dim, y_dim, z_dim;
    T* data;
};
```

9. **Was ändert sich dann beim Zugriff auf die einzelnen Elemente des Arrays?**
   Beim Zugriff auf die einzelnen Elemente des Arrays ändert sich nichts Wesentliches, da die Methode `at` die Berechnung der Position im eindimensionalen Array übernimmt:

```
Image3D<double> img(4, 2, 3);
img.at(1, 1, 1) = 5.0; // Beispielzugriff
```

29.2 Aufgabe 2

```
enum Event {
    Load,
    KeyDown(char),
    Click { x: f32, y: f32 },
}

fn inspect(evt: Event) {
    // ...
}
```

1. **Welche Art von Datentyp ist der Typ `Event`?**
   Der Typ `Event` ist ein benutzerdefinierter Datentyp, genauer gesagt ein `enum` (Enumeration). `Enums` ermöglichen es, einen Typ zu definieren, der mehrere verschiedene, aber festgelegte Werte annehmen kann.
2. **Welche Operationen müssen solche Typen mindestens unterstützen?**
   Solche Typen müssen mindestens folgende Operationen unterstützen:
   - Vergleichsoperationen (z.B. zur Unterscheidung der verschiedenen Varianten des Enums)
   - Zuweisung und Initialisierung (um Variablen dieses Typs zu erstellen und ihnen Werte zuzuweisen)
   - Mustererkennung (`pattern matching`), um je nach Variante des Enums unterschiedliche Operationen auszuführen
3. **Geben Sie eine Implementierung der Funktion `inspect` an, die im Fall eines Click-Events die eingetragenen Koordinaten ausgibt und ansonsten nichts tut.**

```
fn inspect(evt: Event) {
    match evt {
        Event::Click { x, y } => {
            println!("Click event at
       coordinates: ({}, {})", x, y);
        }
        _ => {}
    }
}
```

4. **Was versteht man unter dem Begriff Typinferenz?**
   Typinferenz ist die Fähigkeit des Compilers, den Typ einer Variable automatisch zu bestimmen, ohne dass der Programmierer den Typ explizit angeben muss. Dies ermöglicht kürzeren und oft lesbareren Code.
5. **Erklären Sie anhand eines Beispiels, weshalb Sprachen wie Rust keine implizite Typkonversion erlauben.**
   Rust erlaubt keine implizite Typkonversion, um Fehler zur Kompilierzeit zu vermeiden und die Typensicherheit zu gewährleisten. Zum Beispiel:

```
let x: i32 = 10;
let y: u32 = 20;
// Dieser Code würde in Rust einen Fehler
   verursachen
let z = x + y;
```

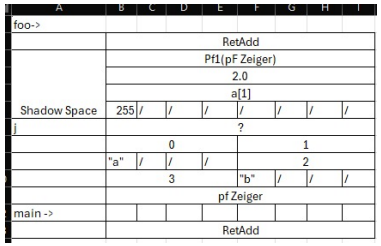
   Hier würde Rust einen Fehler melden, da `i32` und `u32` unterschiedliche Typen sind und keine implizite Konversion zwischen ihnen durchgeführt wird. Dies verhindert unbeabsichtigte Fehler, die durch den Verlust von Daten oder Zeichenfehler entstehen könnten.
6. **Beschreiben Sie den Unterschied zwischen einem Werttyp und einem Referenztyp und geben Sie an, wie man Werttypen von Referenztypen unterscheiden kann.**
   Ein Werttyp speichert die Daten direkt in der Variable. Beispiele sind primitive Datentypen wie `i32` oder `f64`. Ein Referenztyp speichert eine Referenz (Zeiger) auf die Daten, die irgendwo im Speicher liegen. Beispiele sind `&T` oder `Box<T>`. Man kann sie unterscheiden, indem man schaut, ob die Variable selbst die Daten enthält (Werttyp) oder auf die Daten verweist (Referenztyp).
7. **Was ist eine Closure?**
   Eine Closure ist eine anonyme Funktion, die ihren Kontext „einfangen" kann. Das bedeutet, sie kann Variablen, die in ihrem umgebenden Kontext definiert sind, verwenden, ohne dass diese explizit als Parameter übergeben werden müssen. Beispiel:

```
let x = 10;
let add_x = |y| y + x; // `x` wird hier
   eingefangen
println!("{}", add_x(5)); // Ausgabe: 15
```

29.3 Aufgabe 3

```
1   struct A {
2       int i1;
3       int i2;
4       char c;
5   };
6
7   int g = 10;
8
9   int foo (unsigned char u, A const& a1, double d
        , float* f1) {
10      return u + static_cast<int>(d + *f1) + a1.
            i1 + g;
11  }
12
13  int main() {
14      float* pf = new float[3];
15      pf[0] = 2.0f; pf[1] = 0.0f; pf[2] = 42.0f;
16
17      A a[2];
18      a[0].i1 = 0; a[0].i2 = 1; a[0].c = 'a';
19      a[1].i1 = 2; a[1].i2 = 3; a[1].c = 'b';
20
21      int j = foo(255, a[1], 2.0, pf);
22
23      delete[] pf;
24
25      return j;
26  }
27
```

1. **Wie wird die Parameterübergabe der Parameter der Funktion `foo` vom Compiler unter der gegebenen Aufrufkonvention umgesetzt?**
   Unter der Microsoft-x64 Aufrufkonvention werden die ersten vier Parameter in den Registern `rcx`, `rdx`, `r8`, und `r9` übergeben. Die restlichen Parameter werden auf dem Stack übergeben.
   - `unsigned char u` wird in `rcx` übergeben.
   - `A const& a1` wird in `rdx` übergeben.
   - `double d` wird in `r8` übergeben.
   - `float* f1` wird in `r9` übergeben.
2. **Was ist der Rückgabewert von `main`?**
   Der Rückgabewert von `main` ist 271. Die Berechnung erfolgt wie folgt:
   - `u = 255`
   - `d = 2.0`
   - `*f1 = 2.0`
   - `a1.i1 = 2`
   - `g = 10`
   - `return 255 + static_cast<int>(2.0 + 2.0) + 2 + 10`
     `return 255 + 4 + 2 + 10`
     `return 271`
3. **Erklären Sie, weshalb der Ausdruck `*f1` auch ohne die Verwendung des Index-Operators `[]` einen Wert zurückliefert, obwohl an die Funktion doch ein Array übergeben wird.**
   In C++ wird ein Array, das an eine Funktion übergeben wird, als Zeiger auf das erste Element des Arrays behandelt. Der Ausdruck `*f1` dereferenziert diesen Zeiger und liefert den Wert des ersten Elements des Arrays zurück.
4. **In welchem Speicherbereich wird die Variable `g` vom Compiler abgelegt?**
   Die Variable `g` wird im **globalen Datenbereich** (Data Segment) abgelegt, da sie eine globale Variable ist.
5. **Nutzt die aufgerufene Funktion Ressourcen, die eigentlich für die Parameterübergabe vorgesehen sind, muss sie gemäß der Aufrufkonvention die entsprechenden Parameter in den sogenannten Shadow Space kopieren. Dafür muss der Aufrufer diesen Speicherbereich vor dem Aufruf der Funktion bereitstellen. Nehmen Sie im Folgenden an, dass das hier der Fall ist und zeichnen Sie ein Abbild des Stacks und des Heaps des Programms kurz vor der Ausführung der `return`-Anweisung in Zeile 10. Beschriften Sie alle Stackframes, alle lokalen Variablen und ihre Werte sowie eventuell vorhandene Verwaltungsinformation genau. Zeichnen Sie Zeiger als Pfeile. Unbekannte Werte kennzeichnen Sie mit "?".**



Daneben im Heap: die 3 Werte mit jeweils 4 Bytes.

29.4 Aufgabe 4

(e) Aus welchem Grund sind die mit dem C++ Schlüsselwort `union` definierten Datentypen nicht äquivalent zu Rusts `enum`-Typen?

1. **Datenspeicherung:** `union` speichert nur das größte Mitglied und überschreibt andere Mitglieder, während `enum` in Rust die gesamte Struktur speichert und sicherstellt, dass nur eine Variante aktiv ist.
2. **Typensicherheit:** `enum` in Rust garantiert Typensicherheit und erzwingt den Zugriff auf den aktuellen aktiven Datentyp, während `union` in C++ keinen solchen Schutz bietet und es zu undefiniertem Verhalten kommen kann.
3. **Mustererkennung:** `enum` in Rust unterstützt Pattern Matching, das es ermöglicht, sicher zwischen verschiedenen Varianten zu unterscheiden und entsprechend zu handeln. `union` bietet diese Fähigkeit nicht.

29.5 Aufgabe 5

Untersuchen Sie folgendes C++ Programm:

```
1struct Point {
2    float x;
3    float y;
4    void print() {
5        std::cout << "(" << x << "; " << y << ")"
        << std::endl;
6    }
7};
8
9struct PointCompare {
10     bool operator()(Point const& pt1, Point const&
        pt2) {
11         return std::tie(pt1.x, pt1.y) < std::tie(
        pt2.x, pt2.y);
12     }
13};
14
15int main() {
16     std::array<Point, 4> pta {{ {5.0f, 3.2f}, {1.3f
         , 3.4f}, {0, 2}, {1.3f, 2.0f} }};
17     std::sort(pta.begin(), pta.end(), PointCompare
        ());
18     for (auto it = pta.begin(); it != pta.end(); ++
        it) {
19         it->print();
20     }
21     return 0;
22}
```

1. **Was macht das Programm?**
   Das Programm definiert eine Struktur `Point`, die zwei `float`-Mitglieder `x` und `y` hat und eine Methode `print()`, die die Koordinaten ausgibt. Die Struktur `PointCompare` definiert einen Funktionsoperator, der zwei `Point`-Objekte anhand ihrer `x`- und `y`-Werte vergleicht. Im `main()`-Funktionsblock wird ein `std::array` von `Point`-Objekten erstellt und sortiert. Anschließend werden die Punkte in sortierter Reihenfolge ausgegeben.
2. **Was ist die Ausgabe des Programms?**

   ```
   (0; 2)
   (1.3; 2)
   (1.3; 3.4)
   (5; 3.2)
   ```

3. **Welchen Zweck erfüllt die Struktur `PointCompare`?**
   `PointCompare` definiert einen Funktionsoperator, der zwei `Point`-Objekte vergleicht und als Vergleichskriterium für die `std::sort`-Funktion dient.
4. **Seit C++11 gibt es mit Lambda-Ausdrücken eine elegantere Möglichkeit, das gleiche Verhalten zu implementieren. Geben Sie den entsprechenden Lambda-Ausdruck an und nennen Sie die Stelle im Code, an der dieser eingefügt werden muss.**

   ```
   1    std::sort(pta.begin(), pta.end(), [](const
            Point& pt1, const Point& pt2) {
   2        return std::tie(pt1.x, pt1.y) < std::
            tie(pt2.x, pt2.y);
   3    });
   4
   ```

   Der Lambda-Ausdruck wird anstelle von `PointCompare()` in Zeile 18 eingefügt.
5. **Erklären Sie genau, was das Schlüsselwort `auto` in Zeile 20 bewirkt.**
   Das Schlüsselwort `auto` ermöglicht es dem Compiler, den Typ der Variable `it` automatisch zu deduzieren. In diesem Fall ist der Typ `std::array<Point, 4>::iterator`.
6. **Geben Sie den konkreten Typ von `it` aus Zeile 20 an.**
   Der konkrete Typ von `it` ist `std::array<Point, 4>::iterator`.
7. **Modernes C++ verfügt über sogenannte range-based-for-Schleifen (foreach/for(:)). Ersetzen Sie die Schleife in Zeile 20 durch eine solche, die äquivalent zur gegebenen Implementierung ist. Geben Sie hier nur die Schleife aus, nicht den ganzen Code.**

   ```
   1    for (const auto& pt : pta) {
   2        pt.print();
   3    }
   4
   ```

8. **Die range-based-for-Schleifen in C++ werden in der Regel durch eine Compilertransformation in die ursprüngliche Form der Schleife implementiert. Wie nennt man diese Technik, neue Spracheigenschaften durch Transformation in bereits vorhandene zu implementieren?**
   Diese Technik wird als **Desugaring** bezeichnet.
   Desugaring bedeutet, dass eine komplexere Sprachkonstruktion in eine einfachere, fundamentalere Form umgewandelt wird, die der Compiler bereits versteht. Dies vereinfacht die Implementierung und Pflege von Compilern, da neue Features durch Transformationen in bereits existierende Konstrukte umgesetzt werden können. Ein Beispiel für Desugaring ist die Transformation einer range-based-for-Schleife in eine traditionelle for-Schleife:

   ```
   1    for (const auto& pt : pta) {
   2        pt.print();
   3    }
   4
   ```

   wird vom Compiler in etwas Äquivalentes umgewandelt wie:

   ```
   1    for (auto it = pta.begin(); it != pta.end
           (); ++it) {
   2        const auto& pt = *it;
   3        pt.print();
   4    }
   5
   ```

   Diese Transformation stellt sicher, dass neue Sprachkonstruktionen die gleiche Leistungsfähigkeit und Kompatibilität wie bestehende Konstruktionen haben.

29.6 Aufgabe 6

(a) **Nennen Sie die Speicherbereiche, die zur Laufzeit von Programmen im Allgemeinen existieren und geben Sie an, wie diese genutzt werden können (z.B. mittels C++).**
Zur Laufzeit von Programmen existieren im Allgemeinen die folgenden Speicherbereiche:

- **Heap:** Dynamischer Speicherbereich, der für dynamische Speicherallokationen genutzt wird. In C++ erfolgt dies mit `new` und `delete`.
- **Stack:** Speicherbereich für lokale Variablen und Funktionsaufrufe. In C++ werden lokale Variablen automatisch auf dem Stack angelegt.
- **Data Segment:** Bereich für globale und statische Variablen, die zur Laufzeit initialisiert werden. In C++ sind dies Variablen, die außerhalb aller Funktionen definiert sind.
- **BSS Segment:** Bereich für globale und statische Variablen, die nicht initialisiert sind. Diese werden beim Programmstart auf Null gesetzt.
- **Code Segment (Text Segment):** Speicherbereich, der den Maschinencode des Programms enthält.

(b) **Erklären Sie den Begriff Application Binary Interface (ABI).**
Das Application Binary Interface (ABI) ist eine Schnittstelle zwischen zwei Binärmodulen, oft zwischen einer Anwendung und dem Betriebssystem. Es definiert, wie Funktionen aufgerufen werden, welche Register und Speicherbereiche für Parameter und Rückgabewerte verwendet werden, und die Binärlayouts von Datenstrukturen. Ein ABI gewährleistet die Kompatibilität zwischen verschiedenen Compilern und Modulen auf Binärebene.

(c) **Betrachten Sie folgendes C++ Codebeispiel:**

```
1class Base {
2public:
3    Base() : value(0) {}
4    int value;
5};
6
7class Derived : public Base {
8public:
9    Derived() : additionalValue(42) {}
10    int additionalValue;
11};
12
13void printArrayValues(const Base a[], int nElem) {
14    for (int i = 0; i < nElem; ++i) {
15        std::cout << a[i].value << std::endl;
16    }
17}
18
19int main() {
20    Derived d[10];
21    printArrayValues(d, 10);
22    return 0;
23}
```

1. Das Beispiel kompiliert ohne Fehler oder Warnungen, weil ein Array von `Derived` problemlos in eine Funktion übergeben werden kann, die ein Array von `Base` erwartet, da `Derived` von `Base` erbt und daher `Base`-Objekte sind.
2. Die polymorphe Verwendung von Arrays, wie in diesem Beispiel, ist jedoch fehlerhaft, weil der Speicherlayout von `Base` und `Derived` unterschiedlich ist. Das Array `d` besteht aus `Derived`-Objekten, die mehr Speicherplatz benötigen als `Base`-Objekte. Die Funktion `printArrayValues` erwartet jedoch ein Array von `Base`-Objekten und interpretiert die Speicherbereiche fälschlicherweise als `Base`-Objekte. Dies führt zu undefiniertem Verhalten.

(d) **In der Vorlesung haben Sie die Microsoft x86-64 ABI kennengelernt, die Compiler für diese Plattform unterstützen müssen, um zum Beispiel Code für Systemaufrufe erzeugen zu können. Betrachten Sie dazu nun die folgenden beiden Windows-API Funktionssignaturen:**

1. **void* HeapAlloc(void* handle, uint32_t flags, uint64_t size);**
   Um den Aufruf zu dieser Funktion zu erzeugen, muss der Compiler die ersten vier Parameter in die Register `rcx`, `rdx`, `r8` und `r9` laden und anschließend einen `call`-Befehl ausführen. Der Rückgabewert wird in `rax` abgelegt.

   ```
   1    ; rcx: handle
   2    ; rdx: flags
   3    ; r8: size
   4    mov rcx, [handle]
   5    mov rdx, [flags]
   6    mov r8, [size]
   7    call HeapAlloc
   8    mov [result], rax
   9
   ```

2. **void CreateWindow(const char* className, const char* windowName, uint32_t style, int32_t xpos, int32_t ypos, int32_t width, int32_t height, void* parent, void* menu, void* instance, void* userdata);**
   Für diesen Funktionsaufruf muss der Compiler die ersten vier Parameter in die Register `rcx`, `rdx`, `r8` und `r9` laden. Die restlichen Parameter müssen auf den Stack gelegt werden. Anschließend wird ein `call`-Befehl ausgeführt. Der Rückgabewert wird in `rax` abgelegt.

   ```
   1    ; rcx: className
   2    ; rdx: windowName
   3    ; r8: style
   4    ; r9: xpos
   5    sub rsp, 40                ; Stack space for
           the other parameters
   6    mov rcx, [className]
   7    mov rdx, [windowName]
   8    mov r8, [style]
   9    mov r9, [xpos]
   10   mov [rsp+32], ypos         ; Stack parameter
           1
   11   mov [rsp+24], width        ; Stack parameter
           2
   12   mov [rsp+16], height       ; Stack parameter
           3
   13   mov [rsp+8], parent        ; Stack parameter
           4
   14   mov [rsp], menu            ; Stack parameter
           5
   15   mov [rsp-8], instance      ; Stack parameter
           6
   16   mov [rsp-16], userdata     ; Stack parameter
           7
   17   call CreateWindow
   18   add rsp, 40                ; Clean up the
           stack
   19
   ```

29.7 Aufgabe 7

Was versteht man unter einem Referenztyp mit Wertsemantik?
Ein Referenztyp mit Wertsemantik ist ein Datentyp, bei dem Referenzen auf Objekte verwendet werden, aber die Operationen und Vergleiche so gestaltet sind, dass sie wie Werttypen wirken. Das bedeutet, dass Vergleiche und Zuweisungen auf den Inhalt der Objekte (die Werte) und nicht auf die Referenzen selbst abzielen. Beispiele sind Smart Pointer in C++, wie `std::shared_ptr` oder `std::unique_ptr`.

Erklären Sie kurz die folgenden Begriffe:

1. **Converting Cast:**
   Ein Converting Cast ist eine Typumwandlung, bei der der Quelltyp explizit in einen anderen Zieltyp umgewandelt wird, möglicherweise unter Anpassung der darzustellenden Daten. Beispiel in C++: `double d = static_cast<double>(42);`.
2. **Non-converting Cast:**
   Ein Non-converting Cast ist eine Typumwandlung, bei der der Quelltyp in einen Zieltyp umgewandelt wird, der eigentlich der gleiche Typ ist, oder es handelt sich um verschiedene Typen, die aber kompatibel sind, sodass keine Datenanpassung erforderlich ist. Beispiel in C++: `int* p = reinterpret_cast<int*>(voidPtr);`.
3. **Narrowing:**
   Narrowing ist eine Typumwandlung, bei der ein Wert von einem größeren oder präziseren Typ in einen kleineren oder weniger präzisen Typ umgewandelt wird, was zu Datenverlust führen kann. Beispiel: `double d = 3.14; int i = static_cast<int>(d);`.
4. **Widening:**
   Widening ist eine Typumwandlung, bei der ein Wert von einem kleineren oder weniger präzisen Typ in einen größeren oder präziseren Typ umgewandelt wird. Dies führt nicht zu Datenverlust. Beispiel: `int i = 42; double d = static_cast<double>(i);`.
5. **Checked Conversion:**
   Eine Checked Conversion ist eine Typumwandlung, bei der zur Laufzeit überprüft wird, ob die Umwandlung erfolgreich und sinnvoll ist. In C++ kann dies mit `dynamic_cast` durchgeführt werden, wenn zwischen polymorphen Typen umgewandelt wird.
6. **Non-checked Conversion:**
   Eine Non-checked Conversion ist eine Typumwandlung, bei der keine Überprüfung erfolgt, ob die Umwandlung sinnvoll ist. Dies kann zu undefiniertem Verhalten führen, wenn die Umwandlung ungültig ist. Beispiel in C++: `reinterpret_cast`.

Erklären Sie den Begriff Boxing und geben Sie dazu ein Beispiel an.
Boxing ist der Prozess, bei dem ein Werttyp (wie ein primitiver Typ in C# oder Java) in einen Referenztyp umgewandelt wird. Dies ermöglicht, dass Werttypen als Objekte behandelt werden können. Beispiel in C#:

```
1int num = 123;          // Werttyp
2object boxed = num;     // Boxing: num wird in ein
       Objekt umgewandelt
```

(e) **Beschreiben Sie, welche Arten von Gleichheit für programmiersprachliche Objekte in Bezug auf Wert- und Referenztypen existieren.**
Es gibt zwei Hauptarten von Gleichheit:

- **Referenzgleichheit:** Zwei Referenztypen sind gleich, wenn sie auf dasselbe Objekt im Speicher verweisen. In Java wird dies mit dem `==` Operator überprüft.
- **Wertgleichheit:** Zwei Werttypen sind gleich, wenn ihre Inhalte gleich sind. Dies wird durch den `equals()`-Methodenaufruf in Java oder dem Überladen des `==` Operators in C++ erreicht.

29.8 Aufgabe 8

```
#include <iostream>

auto Num(int a) {
    return [a] { return a; };
}

auto Plus(std::function<int()> b, std::function
   <int()> c) {
    return [b, c] { return b() + c(); };
}

int main() {
    auto d = 2;
    auto e = 3;
    auto f = Plus(Num(d), Plus(Num(e), Num(4)))
   ;
    std::cout << f() << std::endl;

    e = 5;
    std::cout << f() << std::endl;

    return 0;
}
```

1. **Was ist** `std::function<int ()>`**?**
   `std::function<int ()>` ist ein Funktions-Wrapper in der C++-Standardbibliothek, der einen beliebigen Callable-Typ speichern kann, der ein `int` zurückgibt und keine Argumente benötigt. Dies kann eine Funktion, ein Funktionsobjekt oder eine Lambda-Funktion sein.
2. **Was macht das Schlüsselwort** `auto`**, das an mehreren Stellen im Programm vorkommt?**
   Das Schlüsselwort `auto` ermöglicht es dem Compiler, den Typ einer Variable automatisch anhand des zugewiesenen Wertes zu deduzieren. Zum Beispiel wird `auto d = 2;` zu `int d = 2;` und `auto f = Plus(...);` wird zu `std::function<int()> f = Plus(...);`.
3. **Was ist die Ausgabe des Programms?**
   Die Ausgabe des Programms ist:

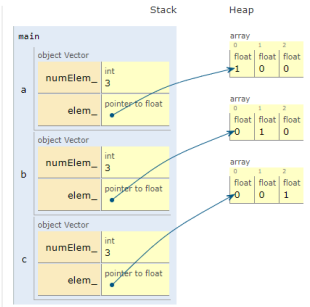
   ```
   9
   9
   ```

   Dies liegt daran, dass die Lambda-Funktionen die Werte von `d` und `e` bei ihrer Erstellung einfangen und speichern. Änderungen an `e` nach der Erstellung der Lambda-Funktion `f` haben keinen Einfluss auf die gespeicherten Werte.
4. **Wie viele Closures werden im Programmablauf insgesamt erzeugt?**
   Im Programmablauf werden insgesamt vier Closures erzeugt:
   - `Num(d)`
   - `Num(e)`
   - `Num(4)`
   - Das Ergebnis von `Plus(Num(e), Num(4))`
5. **Geben Sie an, in welchem Speicherbereich die von den Closures eingefangenen Werte zur Laufzeit des C++-Programms liegen und begründen Sie Ihre Angabe.**
   Die von den Closures eingefangenen Werte liegen im Heap. Lambda-Ausdrücke, die Werte einfangen, speichern diese Werte typischerweise auf dem Heap, da sie möglicherweise länger leben als der Stack-Frame, in dem sie erstellt wurden. Diese Speicherung auf dem Heap ermöglicht es den Closures, ihre eingefangenen Werte auch nach dem Verlassen des Blocks, in dem sie erstellt wurden, beizubehalten.
6. **Nehmen Sie an, es existiert eine Version des Programms in der Programmiersprache JavaScript. In welchem Speicherbereich würden die gefangenen Variablen zur Laufzeit des JavaScript-Programms liegen und warum?**
   In JavaScript würden die gefangenen Variablen im Heap-Speicherbereich liegen. JavaScript verwendet eine Garbage Collection für die Speicherverwaltung, und Closures fangen Variablen aus ihrem lexikalischen Umfeld ein und speichern sie im Heap, um sicherzustellen, dass sie auch nach dem Ende der Funktion, in der sie definiert wurden, zugänglich bleiben.

29.9 Aufgabe 9

```
class Vector {
public:
    Vector(int numElem) : numElem_(numElem) {
        // Aufgabe 4 a)
        elem_ = new float[numElem_];
    }

    ~Vector() {
        // Aufgabe 4 b)
        delete[] elem_;
    }

    float& operator[](int i) {
        // Aufgabe 4 c)
        return elem_[i];
    }

private:
    int numElem_;
    float* elem_;
};

Vector cross(Vector x, Vector y) {
    Vector result(3);
    result[0] = x[1] * y[2] - x[2] * y[1];
    result[1] = x[2] * y[0] - x[0] * y[2];
    result[2] = x[0] * y[1] - x[1] * y[0];

    return result;
}

int main() {
    Vector a(3);
    Vector b(3);

    a[0] = 1; a[1] = 0; a[2] = 0;
    b[0] = 0; b[1] = 1; b[2] = 0;

    Vector c = cross(a, b);
    /*** 1 ***/

    // . . .

    return 0;
}
```

1. **(a) Implementierung des Konstruktors der Klasse Vector (Zeile 4):**
   ```
   Vector(int numElem) : numElem_(numElem) {
           elem_ = new float[numElem_];
   }
   ```
2. **(b) Implementierung des Destruktors der Klasse (Zeile 8):**
   ```
   ~Vector() {
           delete[] elem_;
   }
   ```
   Die Klasse benötigt einen Destruktor, um den dynamisch allokierten Speicher freizugeben, der im Konstruktor mit `new[]` reserviert wurde. Ohne den Destruktor würde es zu Speicherlecks kommen, da der Speicher nicht freigegeben wird.
3. **(c) Implementierung der Methode float& operator[](int i) (Zeile 12):**
   ```
   float& operator[](int i) {
           return elem_[i];
   }
   ```
4. **(e) Erklärung des Absturzes im Debug-Modus unter OS X:**
   Der Absturz mit der Fehlermeldung `error: pointer being freed was not allocated` tritt auf, weil der von der Methode `cross` zurückgegebene `Vector result` als Kopie des `Vector`-Objekts erstellt wird. Dies führt dazu, dass der Kopierkonstruktor von `Vector` verwendet wird, der standardmäßig eine flache Kopie erstellt. Eine flache Kopie bedeutet, dass der Zeiger `elem_` im zurückgegebenen `Vector` auf denselben Speicherbereich zeigt wie der `Vector result` innerhalb der `cross`-Funktion. Wenn der Destruktor für das ursprüngliche `Vector result` aufgerufen wird, wird der Speicher freigegeben, auf den `elem_` zeigt. Später, wenn der Destruktor für die Kopie aufgerufen wird, versucht er, denselben Speicherbereich erneut freizugeben, was zu einem Fehler führt. Um dieses Problem zu beheben, müsste ein tiefer Kopierkonstruktor implementiert werden, der einen neuen Speicherbereich für die Kopie allokiert und die Werte kopiert.



29.10 Aufgabe 10

```
let x: i32 = 42;
let y = 1.0;

x = 1337;

let i: i8 = x;
```

1. **In Zeile 2 des Programms muss der Programmierer den Typ der Variablen** `y` **nicht explizit angeben, obwohl Rust eine statisch und stark typisierte Sprache ist. Geben Sie an, wie man den zugrundeliegenden Mechanismus nennt und erklären Sie, was dabei genau passiert.**
   Der zugrundeliegende Mechanismus wird Typinferenz genannt. Rust verwendet Typinferenz, um den Datentyp einer Variablen basierend auf dem Wert, der ihr zugewiesen wird, automatisch zu bestimmen. In Zeile 2 weist der Wert `1.0` darauf hin, dass `y` den Typ `f64` haben sollte, da Gleitkommazahlen in Rust standardmäßig vom Typ `f64` sind. Der Compiler schließt daher, dass `y` vom Typ `f64` ist.
2. **Die Anweisung in Zeile 4 erzeugt einen Compilerfehler. Erklären Sie, weshalb dieser Fehler entsteht und geben Sie an, welche Änderung man am Programm vornehmen muss, damit die Anweisung erfolgreich kompiliert wird.**
   Der Fehler entsteht, weil Variablen in Rust standardmäßig unveränderlich (`immutable`) sind. Das bedeutet, dass ihr Wert nach der Initialisierung nicht mehr geändert werden kann. Um dies zu ändern, muss die Variable `mutable` gemacht werden, indem das Schlüsselwort `mut` verwendet wird. Die korrekte Zeile 1 lautet:

   ```
   let mut x: i32 = 42;
   ```
3. **Die Anweisung in Zeile 6 führt auch zu einem Compilerfehler. Begründen Sie genau, weshalb der Compiler diese Anweisung nicht zulässt und welche Änderung man nach dem = vornehmen muss, damit die Anweisung erfolgreich kompiliert wird.**
   Der Fehler entsteht, weil Rust keine automatische Typkonvertierung zwischen verschiedenen numerischen Typen zulässt. `x` ist vom Typ `i32`, während `i` vom Typ `i8` ist. Um den Wert von `x` einer `i8`-Variablen zuzuweisen, muss eine explizite Typkonvertierung vorgenommen werden:

   ```
   let i: i8 = x as i8;
   ```

   Das `as`-Schlüsselwort wird verwendet, um die Typkonvertierung durchzuführen.
4. **Was versteht man unter einem algebraischen Datentyp? Erklären Sie, wie sich solche Typen in Rust definieren und verwenden lassen.**
   Ein algebraischer Datentyp (ADT) ist ein Typ, der durch das Kombinieren anderer Typen erstellt wird. Es gibt zwei Hauptarten von ADTs: Summentypen und Produkttypen. Summentypen (auch Varianten oder Enums genannt) können Werte haben, die einer von mehreren möglichen Typen entsprechen. Produkttypen (auch Tupel oder Strukturen genannt) kombinieren mehrere Werte zu einem einzigen Typ.
   - **Definieren eines Summentyps (Enum):**

     ```
     enum Option<T> {
         Some(T),
         None,
     }
     ```
   - **Definieren eines Produkttyps (Struct):**

     ```
     struct Point {
         x: f64,
         y: f64,
     }
     ```
   - **Verwendung:**

     ```
     let some_number = Option::
     Some(5);
     let origin = Point { x: 0.0,
     y: 0.0 };
     ```
5. **Erklären Sie, was man unter dynamischer Typbindung versteht.**
   Dynamische Typbindung bedeutet, dass der Typ einer Variable zur Laufzeit bestimmt wird, nicht zur Kompilierzeit. Dies ermöglicht größere Flexibilität, da der Typ einer Variablen ändern kann, aber auf Kosten der Typsicherheit und möglicherweise der Ausführungsgeschwindigkeit. Rust unterstützt dynamische Typbindung durch das `dyn` Schlüsselwort, meistens in Verbindung mit Trait-Objekten.
6. **In Rust gelten für Referenzen auf Objekte (ausgeliehene Objekte) strenge Regeln, die vom Compiler überwacht werden. Geben Sie an, welche das sind und begründen Sie, weshalb diese sinnvoll sind.**
   Die Regeln für Referenzen in Rust sind:
   - Es kann entweder mehrere unveränderliche Referenzen (`&T`) oder genau eine veränderliche Referenz (`&mut T`) zu einem Objekt geben, jedoch nicht beides gleichzeitig.
   - Die Lebensdauer einer Referenz darf nicht länger sein als die des Objekts, auf das sie verweist.

   Diese Regeln sind sinnvoll, weil sie Datenrennen verhindern und sicherstellen, dass Referenzen immer gültig sind. Dadurch wird Speicher- und Thread-Sicherheit zur Kompilierzeit gewährleistet, ohne dass Laufzeitüberprüfungen erforderlich sind, was zu sicherem und effizientem Code führt.

29.11 Aufgabe 11

1. **Erkären Sie die Begriffe L-Value und R-Value.**
   Ein **L-Value** (Locator Value) ist ein Ausdruck, der eine Speicheradresse bezeichnet. L-Values sind Werte, die auf der linken Seite einer Zuweisung stehen können. Beispiele sind Variablen und Dereferenzen von Zeigern. Ein **R-Value** (Read Value) ist ein Ausdruck, der einen temporären Wert oder einen Wert ohne Speicheradresse bezeichnet. R-Values können auf der rechten Seite einer Zuweisung stehen. Beispiele sind Literale und temporäre Objekte.

```
struct X {
    X(char a_, int b_, short c_) : a(a_), b(b_
       ), c(c_) { };
    char a;
    int b;
    short c;
};

class Y {
public:
    Y() : x(nullptr) { };
    X* x;
};

void init(X** a, Y*& b, Y& c, Y const& d) {
    *a = new X('a', 3, 14);
    b = new Y;
    b->x = new X('b', 2, 71);
    c.x = d.x;
};

int main() {
    Y a;
    Y* b;
    Y c;

    init(&a.x, b, c, a);
    /*** ***/

    return 0;
}
```

2. **(a) Speicherlayout der Struktur X und der Klasse Y:**
   Struktur X:
   - **char a:** 1 Byte
   - **Padding:** 3 Bytes (um das Alignment von 4 Bytes für den nächsten `int` zu erreichen)
   - **int b:** 4 Bytes
   - **short c:** 2 Bytes
   - **Padding:** 2 Bytes (um das Alignment von 4 Bytes für die gesamte Struktur zu erreichen)

   Gesamtgröße: 12 Bytes (inkl. 5 Bytes Padding)
   Klasse Y:
   - **X* x:** 4 Bytes

   Gesamtgröße: 4 Bytes (kein Padding erforderlich)
3. **(b) Untersuchung der Funktion init():**
   Parameter:
   - `X** a`: Zeiger auf einen Zeiger auf ein `X` (call-by-value)
   - `Y*& b`: Referenz auf einen Zeiger auf ein `Y` (call-by-reference)
   - `Y& c`: Referenz auf ein `Y` (call-by-reference)
   - `Y const& d`: Konstante Referenz auf ein `Y` (call-by-reference)
4. **(c) Zeichnen des Stacks und des Heaps an der Stelle /* *** */:**

   Stack:

Variable	Address	Value
main() - return address	0x0000	0x1234
main() - a	0x0004	0x1000
main() - b	0x0008	0x2000
main() - c	0x000C	0x3000
init() - return address	0x0010	0x1234
init() - a	0x0014	0x0004
init() - b	0x0018	0x0008
init() - c	0x001C	0x000C
init() - d	0x0020	0x0004

   Heap:

Address	Value
0x1000	X('a', 3, 14)
0x2000	Y()
0x2004	X('b', 2, 71)
0x3000	Y()

5. **(d) Speicherfreigabe vor** `return 0;`**:**
   Vor `return 0;` in `main()` muss der angeforderte Speicher freigegeben werden:

   ```
   delete a.x;
   delete b->x;
   delete b;
   ```

29.12 Aufgabe 12

1. **Beschreiben Sie die grundlegenden Konzepte, auf denen die objektorientierte Programmierung basiert.**
   Die objektorientierte Programmierung (OOP) basiert auf vier grundlegenden Konzepten:
   - **Kapselung:** Die Kombination von Daten und den Methoden, die auf diese Daten zugreifen, in einer einzigen Einheit, der Klasse. Dies schützt die Daten vor unberechtigtem Zugriff und Modifikation.
   - **Abstraktion:** Die Fähigkeit, komplexe Systeme durch einfachere Modelle zu repräsentieren, indem nur die wesentlichen Details dargestellt und unwesentliche Details verborgen werden.
   - **Vererbung:** Die Möglichkeit, neue Klassen auf Basis bestehender Klassen zu erstellen, wodurch Code-Wiederverwendung ermöglicht und Hierarchien von Klassen definiert werden.
   - **Polymorphismus:** Die Fähigkeit, dass unterschiedliche Klassen durch die gleiche Schnittstelle angesprochen werden können, wodurch Flexibilität und Erweiterbarkeit des Codes verbessert werden.
2. **Erklären Sie den Unterschied zwischen einem Upcast und einem Downcast innerhalb einer objektorientierten Klassenhierarchie.**
   Upcast: Ein Upcast ist das Casten eines Objekts von einer abgeleiteten Klasse zu einer Basisklasse. Dies ist immer sicher, da die abgeleitete Klasse alle Eigenschaften der Basisklasse erbt. Beispiel:

   ```
   Derived* derived = new Derived();
   Base* base = derived;  // Upcast
   ```

   Downcast: Ein Downcast ist das Casten eines Objekts von einer Basisklasse zu einer abgeleiteten Klasse. Dies kann unsicher sein, da die Basisklasse nicht notwendigerweise alle Eigenschaften der abgeleiteten Klasse hat. Beispiel:

```
Base* base = new Derived();
Derived* derived = static_cast<Derived*>(
    base);  // Downcast
```

3. **Die .NET Runtime führt Laufzeitüberprüfungen bei einem Downcast aus. Ist ein Downcast innerhalb einer Klassenhierarchie ungültig, wird dies durch eine Ausnahme angezeigt. Finden ähnliche Prüfungen auch in C++ statt? Welche Möglichkeiten zur dynamischen Typprüfung existieren in C++?**
   In C++ können ähnliche Prüfungen mittels `dynamic_cast` durchgeführt werden. Wenn der Downcast ungültig ist und `dynamic_cast` verwendet wird, wird der Cast fehlschlagen und `nullptr` zurückgeben (bei Zeigern) oder eine `bad_cast`-Ausnahme werfen (bei Referenzen).

```
Base* base = new Derived();
Derived* derived = dynamic_cast<Derived*>(
    base);
if (derived == nullptr) {
    // Cast failed
}
```

4. **Erläutern Sie, wie Untertyp-Polymorphismus in objektorientierten Programmiersprachen implementiert wird.**
   Untertyp-Polymorphismus wird durch Vererbung und virtuelle Funktionen implementiert. Eine Basisklasse definiert eine Schnittstelle mittels virtueller Funktionen, die von abgeleiteten Klassen überschrieben werden können. Ein Zeiger oder eine Referenz auf die Basisklasse kann verwendet werden, um Objekte der abgeleiteten Klasse anzusprechen und die korrekte Funktion wird zur Laufzeit (dynamisch) aufgerufen.

```
class Base {
public:
    virtual void print() { std::cout << "
    Base" << std::endl; }
};

class Derived : public Base {
public:
    void print() override { std::cout << "
    Derived" << std::endl; }
};

Base* b = new Derived();
b->print();  // Outputs "Derived"
```

5. **Nennen Sie andere Arten von Polymorphismus. Geben Sie jeweils ein Beispiel an.**
   Ad-hoc-Polymorphismus: Auch bekannt als Überladung. Dies tritt auf, wenn mehrere Funktionen denselben Namen, aber unterschiedliche Parameterlisten haben.

```
void print(int i) { std::cout << i << std
    ::endl; }
void print(double d) { std::cout << d <<
    std::endl; }
```

   Parametrischer Polymorphismus: Auch bekannt als Generika. Dies tritt auf, wenn Funktionen oder Klassen so geschrieben werden, dass sie mit beliebigen Typen arbeiten können.

```
template <typename T>
void print(T value) { std::cout << value
    << std::endl; }
```

6. **Erklären Sie den Begriff Duck-Typing. Geben Sie ein Beispiel in einer Programmiersprache Ihrer Wahl an.**
   Duck-Typing ist ein Konzept, bei dem die Typenprüfung zur Laufzeit erfolgt und basiert auf dem Verhalten (Methoden und Eigenschaften) eines Objekts und nicht auf seiner expliziten Typdefinition. Wenn ein Objekt alle erforderlichen Methoden hat, kann es verwendet werden, unabhängig von seinem Typ. Beispiel in Python:

```
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I am quacking like a duck!"
    )

def make_it_quack(duck):
    duck.quack()

d = Duck()
p = Person()
make_it_quack(d)  # Outputs "Quack!"
make_it_quack(p)  # Outputs "I am quacking
    like a duck!"
```

29.13 Aufgabe 13

1. **Welches sind die grundlegenden faktischen Anforderungen an eine Programmiersprache? Was muss sie erfüllen?**
   Eine Programmiersprache muss mehrere grundlegende Anforderungen erfüllen:
   - **Syntax:** Klar definierte Regeln für die Struktur und das Format von Anweisungen und Ausdrücken.
   - **Semantik:** Bedeutungsvolle Operationen, die den Anweisungen entsprechen und definieren, wie Programme ausgeführt werden.
   - **Abstraktion:** Möglichkeiten zur Abstraktion von Details durch Funktionen, Klassen, Module etc.
   - **Typensystem:** Ein System zur Definition und Überprüfung von Datentypen, das Typsicherheit gewährleistet.
   - **Standardbibliotheken:** Umfangreiche Bibliotheken, die häufig benötigte Funktionen bereitstellen.
   - **Portabilität:** Die Fähigkeit, auf verschiedenen Plattformen und Systemen ausgeführt zu werden.
2. **Wie werden Programmiersprachen klassifiziert? Nennen Sie für jede Klasse ein Beispiel.**
   Programmiersprachen können in verschiedene Klassen unterteilt werden:
   - **Imperativ:** Sprachen, die auf Anweisungen basieren, die den Computer explizit anweisen, was zu tun ist. Beispiel: C.
   - **Deklarativ:** Sprachen, die spezifizieren, was das Programm tun soll, ohne explizit anzugeben, wie es getan werden soll. Beispiel: SQL.
   - **Objektorientiert:** Sprachen, die auf Objekten und Klassen basieren. Beispiel: Java.
   - **Funktional:** Sprachen, die auf mathematischen Funktionen basieren. Beispiel: Haskell.
   - **Logisch:** Sprachen, die auf formaler Logik basieren. Beispiel: Prolog.
3. **Was versteht man unter dem Begriff „Gültigkeitsbereich"?**
   Der Gültigkeitsbereich (Scope) bezeichnet den Bereich im Programm, in dem eine Variable oder Funktion sichtbar und zugänglich ist. Es gibt verschiedene Arten von Gültigkeitsbereichen, wie globaler, lokaler und Block-Gültigkeitsbereich, die definieren, wo Variablen und Funktionen innerhalb des Programms verwendet werden können.
4. **Nennen und erklären Sie knapp die möglichen Speicherallokationsmechanismen für programmiersprachliche Objekte.**
   - **Statische Allokation:** Speicher wird zur Compile-Zeit reserviert und bleibt während der gesamten Programmlaufzeit bestehen. Beispiel: globale Variablen.
   - **Automatische Allokation:** Speicher wird zur Laufzeit beim Eintritt in einen Block reserviert und beim Verlassen des Blocks freigegeben. Beispiel: lokale Variablen.
   - **Dynamische Allokation:** Speicher wird zur Laufzeit explizit angefordert und freigegeben. Beispiel: Speicher, der mit `malloc` in C oder `new` in C++ reserviert wird.
5. **Erklären Sie die Begriffe „referential opacity" und „referential transparency".**
   Referential Transparency: Ein Ausdruck ist referentiell transparent, wenn er durch seinen Wert ersetzt werden kann, ohne das Verhalten des Programms zu ändern. Beispiel: `f(x) = x + 2` ist referentiell transparent, weil `f(3)` immer 5 ergibt. **Referential Opacity:** Ein Ausdruck ist referentiell opak, wenn er nicht durch seinen Wert ersetzt werden kann, ohne das Verhalten des Programms zu ändern. Beispiel: `rand()` ist referentiell opak, weil `rand()` jedes Mal einen anderen Wert zurückgibt.
6. **Welche Arten von „Gleichheit" existieren für programmiersprachliche Objekte?**
   - **Strukturelle Gleichheit:** Zwei Objekte sind strukturell gleich, wenn ihre Datenstruktur und ihre Inhalte gleich sind.
   - **Referenzielle Gleichheit:** Zwei Objekte sind referenziell gleich, wenn sie dieselbe Speicheradresse haben.
7. **Was ist eine „virtual table"?**
   Eine „virtual table" (vtable) ist eine Datenstruktur, die von vielen objektorientierten Programmiersprachen zur Unterstützung des dynamischen Dispatch verwendet wird. Sie enthält Zeiger auf die virtuellen Funktionen einer Klasse. Wenn eine virtuelle Methode auf einem Objekt aufgerufen wird, verwendet der Compiler die vtable des Objekts, um die richtige Methode zu finden und aufzurufen. Dies ermöglicht Polymorphismus und dynamisches Binden von Methodenaufrufen.

29.14 Aufgabe 14

```
#include <functional>
#include <iostream>

template <typename T>
std::function<T (T)> makeConverter(T factor, T
    offset) {
    return [=] (T input) -> T { return (offset
    + input) * factor; };
}

int main() {
    auto milesToKm = makeConverter(1.60936,
    0.0);
    auto poundsToKg = makeConverter(0.45460,
    0.0);
    auto fahrenheitToCelsius = makeConverter
    (0.5556, -32.0);

    std::cerr << milesToKm(10) << std::endl;
    std::cerr << poundsToKg(2.5) << std::endl;
    std::cerr << fahrenheitToCelsius(98) << std
    ::endl;

    return 0;
}
```

1. **Was ist die Ausgabe des Programmes?**
   Die Ausgabe des Programms ist:

```
16.0936
1.1365
36.0888
```

2. **An einer Stelle des Programms kommt eine anonyme Funktion vor. Geben Sie das Programmstück an.**
   Das Programmstück mit der anonymen Funktion (Lambda-Funktion) ist:

```
return [=] (T input) -> T { return (
    offset + input) * factor; };
```

3. **Was ist „[=] (T input) -> T ... "? Wozu dient die Zeichenfolge „[=]"? Was würde passieren, wenn man hier stattdessen „[ ]" angeben würde?**
   „[=] (T input) -> T ... " ist eine Lambda-Funktion in C++. **„[=]"** ist ein Lambda-Capture-Ausdruck, der angibt, dass alle in der äußeren Funktion sichtbaren Variablen (in diesem Fall `factor` und `offset`) per Wert (`by value`) in die Lambda-Funktion kopiert werden sollen. Dies bedeutet, dass die Lambda-Funktion eine Kopie dieser Variablen erhält und sie innerhalb der Lambda-Funktion verwendet werden können. Wenn man stattdessen „[ ]" angeben würde, würde die Lambda-Funktion keine äußeren Variablen einfangen können. Das bedeutet, dass `factor` und `offset` innerhalb der Lambda-Funktion nicht zugänglich wären, und der Compiler würde einen Fehler melden, da diese Variablen in der Lambda-Funktion nicht bekannt sind.

29.15 Aufgabe 15

```
#include <cstring>

template <typename T>
class Matrix {
public:
    Matrix(int n, int m) : n_(n), m_(m) {
        elem_ = new T*[n_];
        for (int i = 0; i < n_; ++i) {
            elem_[i] = new T[m_];
            memset(elem_[i], 0, sizeof(T) * m_)
    ; /* Alle Elemente einer Zeile -> 0 */
        }
    }

    ~Matrix() {
        for (int i = 0; i < n_; ++i) {
            delete[] elem_[i];
        }
        delete[] elem_;
    }

    T* operator[](int i) {
        return elem_[i];
    }

private:
    int n_, m_;
    T** elem_;
};
```

1. **Begründung für den Destruktor:**
   Der Destruktor der Klasse `Matrix` war ursprünglich leer, was falsch ist, da der in der Klasse dynamisch allokierte Speicher (die 2D-Array `elem_`) nicht freigegeben wird. Dies führt zu Speicherlecks, weil der allokierte Speicher nach der Zerstörung der `Matrix`-Objekte nicht zurückgegeben wird.
   Richtige Implementierung des Destruktors:
   Der korrekte Destruktor iteriert über alle Zeilen des 2D-Arrays und gibt den Speicher jeder Zeile frei, bevor er den Speicher für das Zeiger-Array selbst freigibt s.o.
2. **Begründung für die Zugriffsmethode:**
   Die Klasse `Matrix` ist ohne Zugriffsmethoden weitgehend unbrauchbar, da es keine Möglichkeit gibt, auf die einzelnen Elemente der Matrix zuzugreifen. Durch Überladen des `operator[]` können wir einen bequemen Zugriff auf die Elemente der Matrix ermöglichen.
   Implementierung der Zugriffsmethode:
   Der `operator[]` ermöglicht den Zugriff auf eine bestimmte Zeile der Matrix T* operator