universität wien

Wichtige Informationen zu den Collaborate Aufzeichnungen

- Vorbesprechung, Vorlesungen und Repetitorien werden aufgezeichnet.
  - Fragen können nur per öffentlichem Chat gestellt werden, dieser wird in den Aufzeichnungen anonymisiert.
  - Vermeiden Sie die Bekanntgabe privater Daten im Chat (auch Namensnennungen wie z.B. @Franz Meier)!
- Übungseinheiten werden eventuell auch aufgezeichnet.
  - Fragen können auch per Mikrofon gestellt werden.
  - Verwenden Sie das Mikrofon nicht, wenn Sie mit der Aufzeichnung Ihrer Stimme und/oder der anschließenden Veröffentlichung nicht einverstanden sind!
  - Vermeiden Sie die Bekanntgabe privater Daten im Chat!
  - Breakout Räume zur intensiven Einzelbetreuung werden nicht aufgezeichnet.
- Tutorien werden nicht aufgezeichnet.

universität wien

Programmierung 1

Vorlesung

Peter Beran,
Reinhold Dunkl,
Conrad Indiono,
Sonja Kabicher-Fuchs,
Gerd Krislaty,
Beate Scheibel,
Manfred Schüttengruber,
Sebastian Schrittwieser,
Georg Sonneck,
Helmut Wanek (Vortragender)

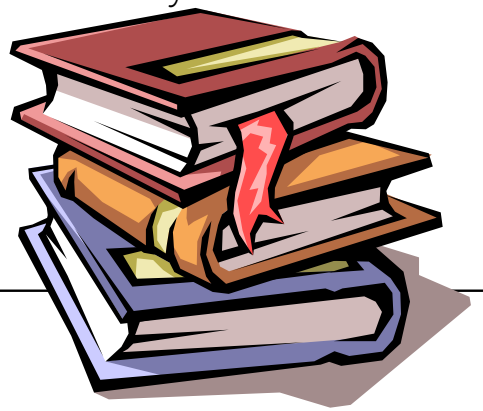
Basierend auf Bjarne Stroustrup: „Programming Principles and Practice Using C++“

mit Beiträgen von C. Bruckmann, M. Hitz, T. Mück und E. Schikuta

universität wien

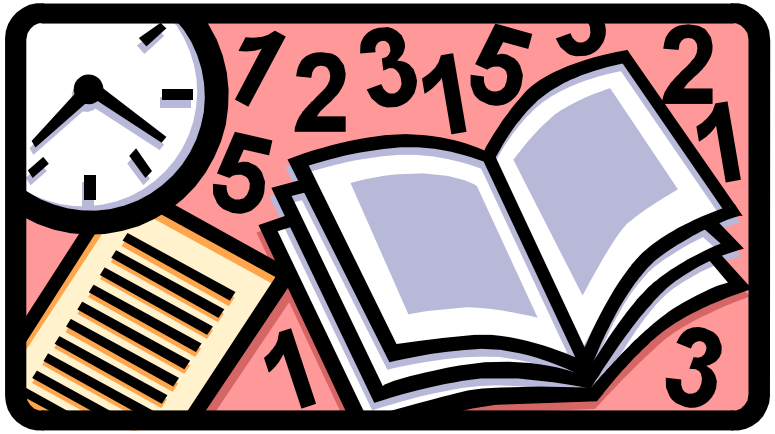
Literatur

- H.M. Deitel und P.J. Deitel: C++ How to Program. Prentice Hall
- H.M. Deitel & P.J. Deitel & T.R. Nieto: C++ in the Lab, Prentice Hall (Übungsbuch zu C++ How to Program)
- Bjarne Stroustrup: Einführung in die Programmierung mit C++, Pearson Studium.
- Bjarne Stroustrup: Die C++ Programmiersprache. Addison Wesley.
- Bjarne Stroustrup: Programming Principles and Practice Using C++, Addison Wesley.



Folien des Vortrags

- Online Präsentation im Web
  - erreichbar über die Homepage
  - https://cewebs.cs.univie.ac.at/PR1/ws21
  - enthält zusätzlich alle dynamischen Folien (Symbol)



universität wien

1. Grundlagen

universität wien

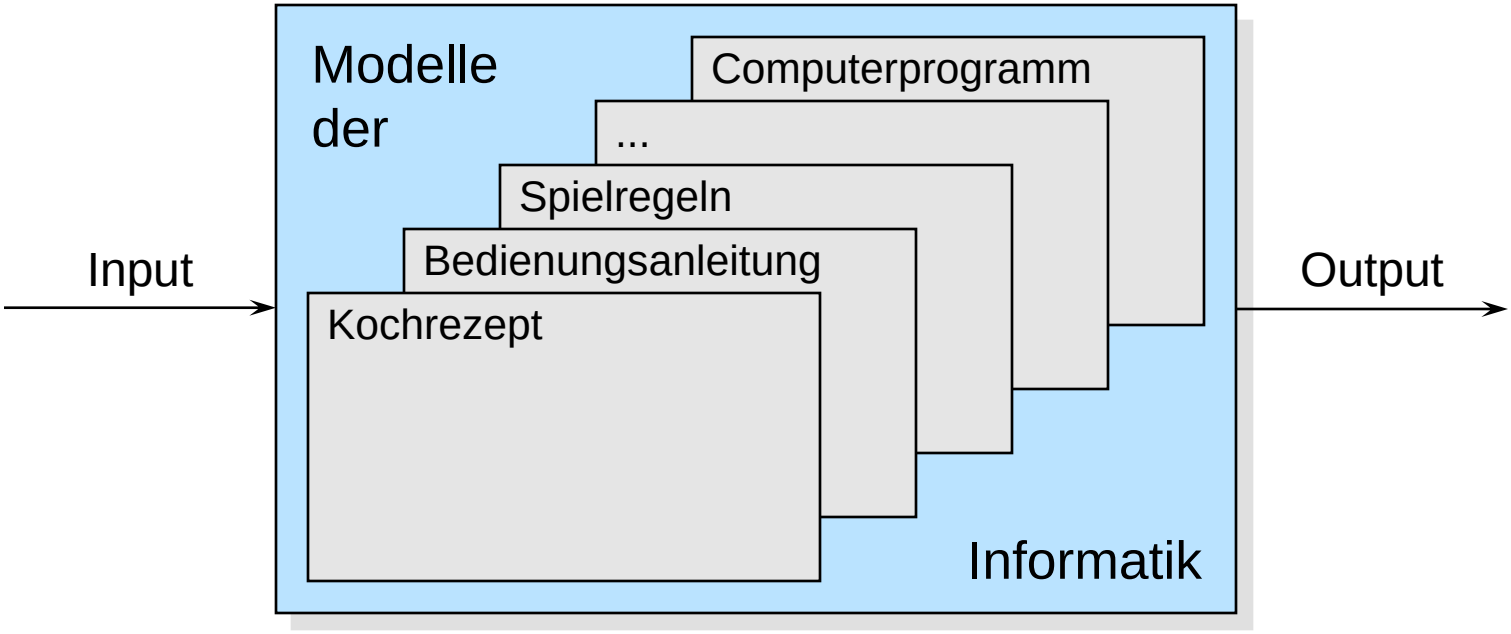
Was ist Informatik (informatics / computer science)

• Die Wissenschaft der *Informatik* umfasst alle Modelle (Methoden, Verfahren, Konzepte, etc.), die dazu dienen, eine gegebene Eingabe in eine beliebige (gewünschte) Ausgabe zu verwandeln

Fasst man Input und Output als (Bit)Muster (pattern) auf, so befasst sich die Informatik mit der Umwandlung von Mustern und ist damit eng mit der Mathematik verknüpft („mathematics as a science of patterns“).

universität wien

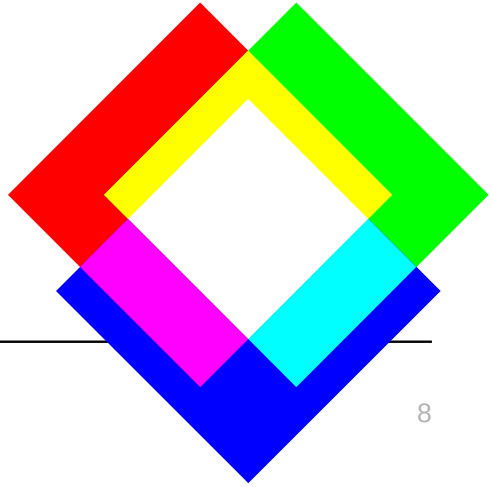
Modelle der Informatik



Beispiele für Modelle

- Kochrezept als Folge von Anweisungen
  - Amerikanischer Wildreis
    - 1 Tasse ergibt 3 Portionen
    - Reis gründlich waschen
    - 1 Tasse Reis in 3 Tassen kochendes Wasser geben
    - kurz aufkochen lassen
    - bei schwacher Hitze 25 min bedeckt dünsten
    - Reis abdecken, salzen, mit Gabel auflockern
    - restliche Flüssigkeit verdampfen

- Beschreibung von Ursache-Wirkungszusammenhängen, von Verfahren mit bekannten Folgen
  - Weißes Licht erhält man,
    wenn man rotes, grünes und blaues Licht mischt



universität wien

Algorithmus (algorithm)

Unter Algorithmus versteht man die schrittweise Vorschrift zur Berechnung gesuchter aus gegebenen Größen, in der jeder Schritt aus einer Anzahl eindeutig ausführbarer Operationen und einer Angabe über den nächsten Schritt besteht.

- Ursprung
  - Algorithmus » Berechnungsvorschrift «
  - Ben Musa Al-Chwarizmi (usbekischer Mathematiker um 825), erstes Buch über Algebra
  - arithmos ... griechisches Wort für Zahl

universität wien

Algorithmus - Eigenschaften

- Eingangswerte/Ausgabewerte (input/output)
  - EW sind vor, AW nach der Ausführung bekannt
- Eindeutigkeit (unambiguity)
  - Jeder Schritt der Ausführung muss eindeutig sein, keine Mehrdeutigkeiten möglich
- Effektivität (effectivity)
  - Jeder Schritt muss ausführbar sein
- Determinismus (determinism)
  - Zufall spielt keine Rolle, dieselben Eingangswerte liefern immer wieder dieselben Ausgangswerte
- Endlichkeit (finiteness)
  - Statisch: mit endlich vielen Zeichen formulierbar
  - Dynamisch: in endlich vielen Schritten beendbar
- Vollständigkeit (completeness)
  - Sollte vollständig sein, sollte alle möglichen Fälle behandeln
- Korrektheit (correctness)
  - Sollte das gewünschte Ergebnis liefern

universität wien

Algorithmus - Definition

- Der Begriff des Algorithmus wird durchaus unterschiedlich definiert

- Manchmal ist es hilfreich, auf eine der zuvor beschriebenen Eigenschaften zu verzichten, um bestimmte Probleme überhaupt, oder effizient lösen zu können
  - Z.B.: indeterministische Algorithmen
    - –Monte Carlo Algorithmen: Liefern (selten) falsche Ergebnisse (nicht korrekt, indeterministisch)
    - –Las Vegas Algorithmen: Terminieren nicht immer, liefern aber, falls sie terminieren, das korrekte Ergebnis (nicht finit, indeterministisch)

- In dieser LV werden wir nur Algorithmen betrachten, die alle der zuvor beschriebenen Eigenschaften besitzen

universität wien

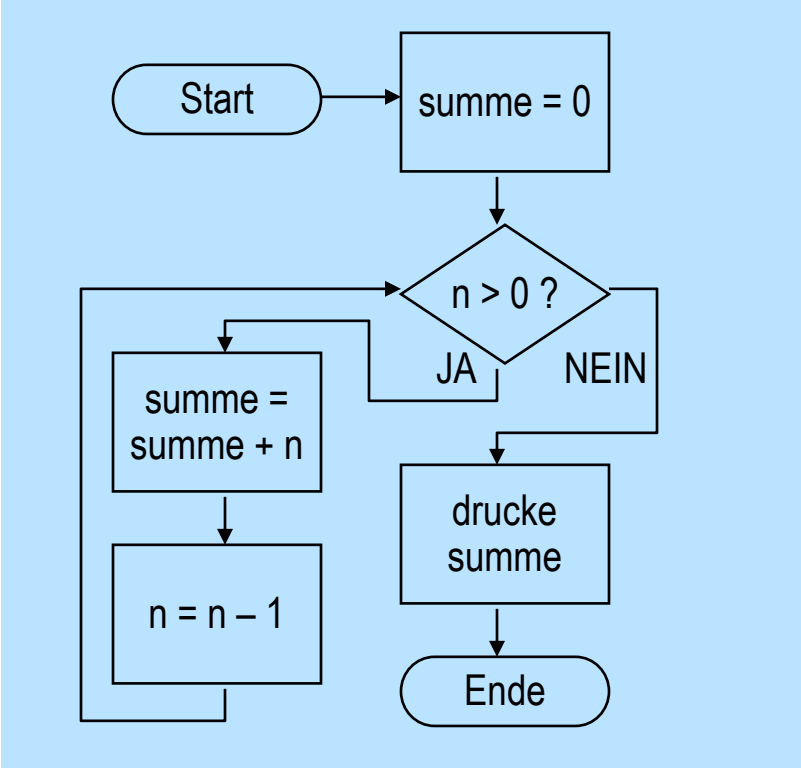
Algorithmendarstellung

- Verbreitete Beschreibungsmethoden für Algorithmen

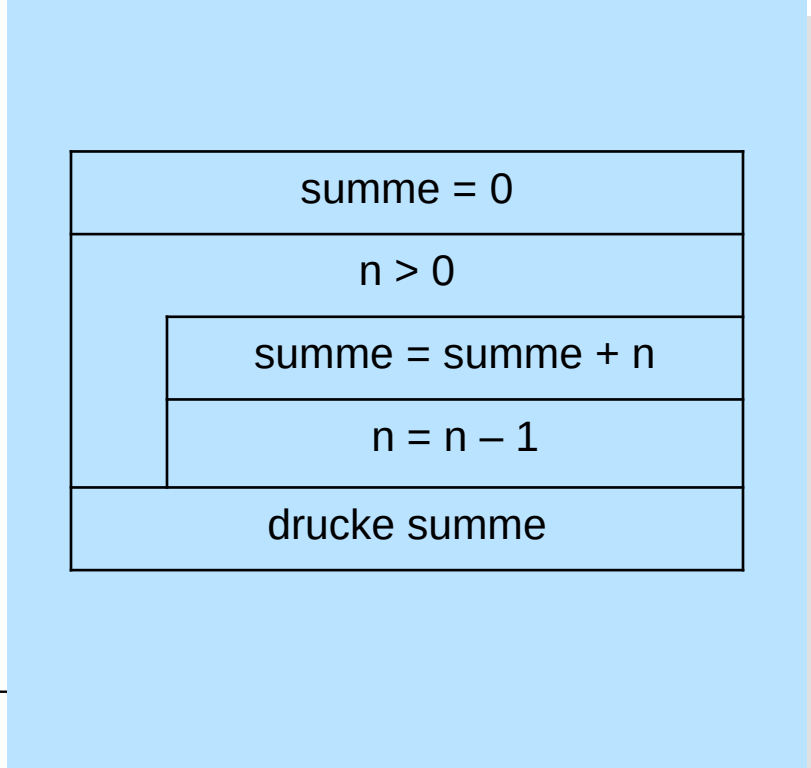
- Graphisch
  - Ablaufdiagramme
  - Struktogramme
  - UML-Diagramme
- Textuell
  - Natürliche Sprache (eventuell auch mathematische Formalismen)
  - Pseudocode
  - Programmiersprachen

Darstellung – graphisch (1)

Flussdiagramm (flow chart)
- Anweisungen stehen in Knoten
- Kontrollfluss: gerichtete Kanten



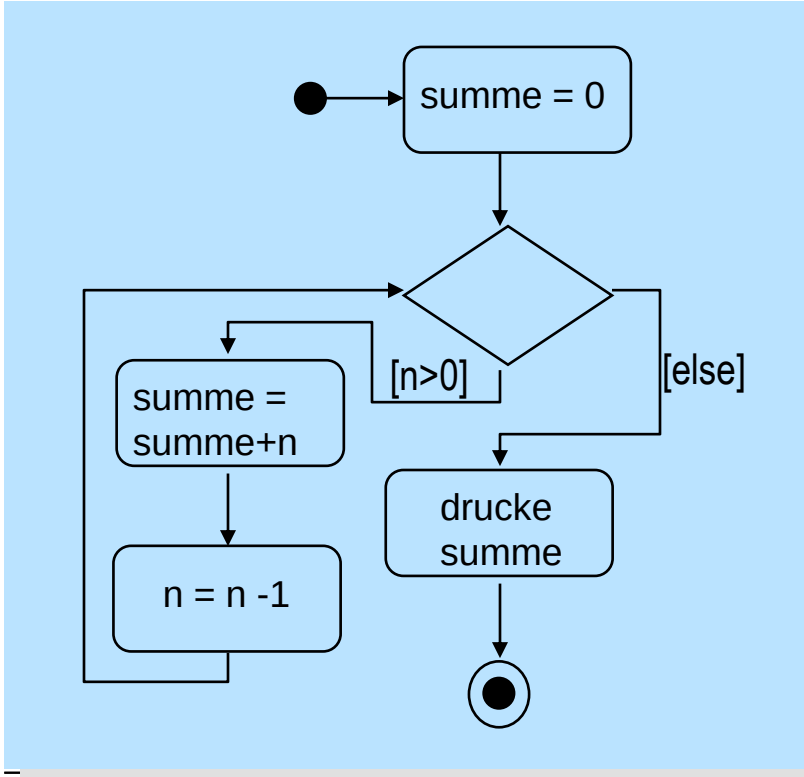
Struktogramm (structogram, Nassi-Shneiderman diagram)
- Anweisungen stehen in Blöcken
- Kontrollfluss: Form, Struktur der Blöcke



Darstellung – graphisch (2)

UML - Diagramm (UML diagram)

Ähnlich wie Flussdiagramm



Eigenschaften der graphischen Darstellung:

Pro:

- leicht erfassbar
- Standardnotation
- auch dem Anwender verständlich (Diskussionsbasis)

Con:

- große Programme unüberschaubar
- schwierig direkt in Programmiersprache umsetzbar
- schwer editierbar (bzw. müssen spezielle Tools eingesetzt werden)
- automatische Codegenerierung eher beschränkt (meist nur Prozedurköpfe)

universität wien

Darstellung – natürliche Sprache

Beginne mit einer leeren Summe. Solange n größer 0 ist, addiere n zur Summe und vermindere n um 1.

Pro:

Für Menschen direkt verständlich

Con:

Keine Möglichkeit der Automatisierung

Komplexe Sachverhalte haben meist eine lange, umständliche Beschreibung (evtl. eigener Formalismus nötig)

universität wien

Darstellung – Pseudocode

Programmiersprachenderivate

Anlehnung an eine Programmiersprache (z. B. Pascal, MODULA-2);
Ziel, eine, der natürlichen Sprache möglichst nahe, Kunstsprache zu schaffen (standardisierte Darstellung!)

```
begin comment Das ist ALGOL;
      integer n,i,summe;
      read(n);
      summe := 0;
      for i := n step -1 until 1
         do summe := summe + i;
      print(summe)
end
```

Stilisierte Prosa

Beschreibung der schrittweisen Ausführung

```
Schritt 1:Initialisiere.
          Setze summe auf 0.
Schritt 2:Abarbeitung der Schleife.
          Solange n größer als 0,
          addiere n zu summe,
          ziehe 1 von n ab.
Schritt 3:Ausgabe.
          Drucke summe.
```

Pro: Programmiersprache sehr ähnlich, direkte Umsetzung einfach

Con: ähnliche Komplexität wie Programmiersprache, für den Anwender oft schwer verständlich

universität wien

Programmiersprache (programming language)

- Sprache mit streng formaler Syntax (“Grammatikregeln”) und Semantik (“Bedeutung”) zur Formulierung von Programmen.

- Bietet für Menschen die Möglichkeit, den Algorithmus zu beschreiben, zu verstehen und weiterzugeben.

- Kann automatisch in exekutierbaren Maschinencode umgewandelt werden.

universität wien

Programm (program) allgemein

- Ein Programm ist eine syntaktisch korrekt geformte (wohlgeformte) Formel einer Programmiersprache.

- Ein Programm erlaubt die Berechnung der Ausgangsdaten aus den Eingangsdaten mit Hilfe des Computers.

- Ein deterministisches Programm kann als eine mathematische Funktion betrachtet werden, die die Menge der möglichen Eingaben in die Menge der möglichen Ausgaben abbildet ($\boldsymbol{f}: \boldsymbol{E} \rightarrow \boldsymbol{A}$) und mit Hilfe des Computers berechnet werden kann

universität wien

Programm (in dieser Lehrveranstaltung)

- Im Kontext dieser Lehrveranstaltung verstehen wir unter einem Programm die Beschreibung eines Algorithmus, die es ermöglicht, diesen Algorithmus durch einen Computer ausführen zu lassen.

- Ein Programm ist die Beschreibung eines Algorithmus in einer Programmiersprache. (Quelltext; source code)

- Ein Programm ist die Implementierung eines Algorithmus auf einem Computer. (Maschinencode; executable code)

universität wien

Summenberechnung als C++ - Programm

```
#include <iostream>
using namespace std;

int main() {
  int n;
  cin >> n;
  int summe {0};
  while(n > 0) {
    summe += n;
    --n;
  }
  cout << summe << '\n';
  return 0;
}
```

Compiler →

← Decompiler

```
01111111010001010100110001
00011000000010000000010000
00010000000000000000000000
00000000000000000000000000
00000000000000000000000000
00000100000000001111100000
00000000000100000000000000
00000000000000000000000000
00000000000000000000000000
00000000000000000000000000
00000000000000000000000000
00000000000000000000000000
00000001110100000000001000
00000000000000000000000000
0000… (2952 bytes)
```

universität wien

Alternativen zur Kompilation

- Statt den gesamten Quellcode auf einmal zu übersetzen, kann der Code (etwa während des Eintippens) Stück für Stück (meist zeilenweise) interpretiert und ausgeführt werden.
- Das Programm, das diese Umsetzung vornimmt, heißt Interpreter.
- Verschiedene Programmiersprachen eignen sich unterschiedlich gut für Interpreter bzw. Compiler.
- Auch die Kompilierung in einen Zwischencode, der dann interpretativ abgearbeitet wird, ist gebräuchlich.
- Vgl. Simultandolmetscher / Übersetzung eines gesamten Werkes

universität wien

Binder (Linker)

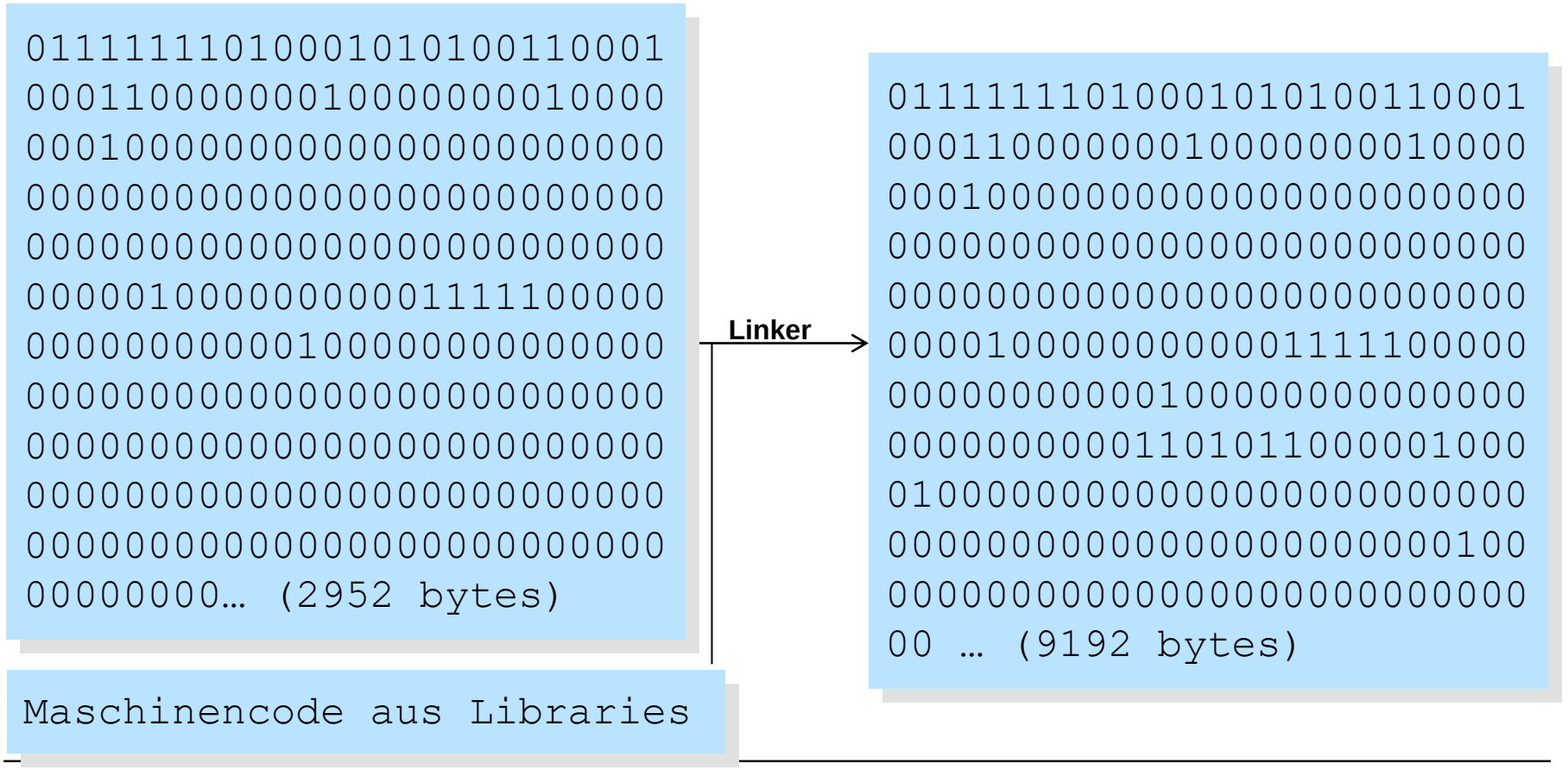
- In der Regel wird nicht immer der gesamte Programmtext auf ein Mal übersetzt.
- Häufig verwendete Programmteile (z.B. die I/O-Funktionalität) werden z.B. in Libraries ausgelagert, die bereits in kompilierter Version zur Verfügung stehen.

- Aufgabe des Linkers ist es, die unabhängig übersetzten Teilprogramme zu einem exekutierbaren Gesamtprogramm zusammenzusetzen.

- (Alternativ können fehlende Programmteile auch erst bei Bedarf zur Laufzeit an das Programm gebunden werden – Dynamic Link Library)

universität wien

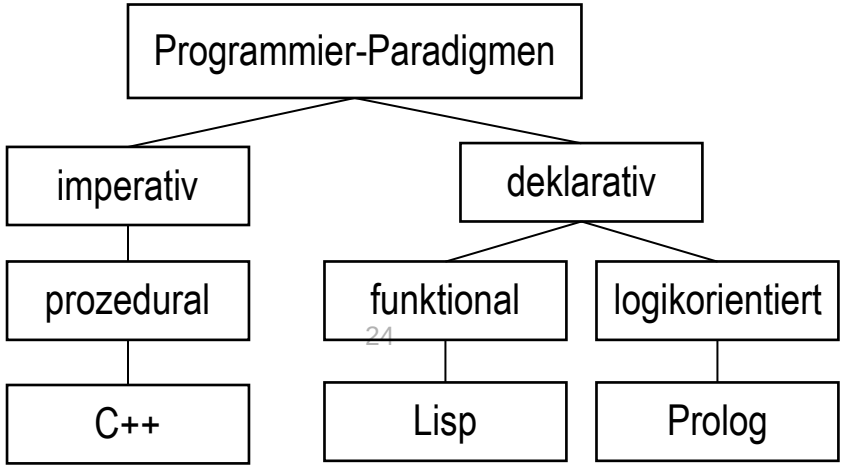
Summenberechnung als C++ - Programm



universität wien

Programmier-Paradigmen (programming paradigms)

- Paradigma
  - „ … Das, was den Mitgliedern einer wissenschaftlichen Gemeinschaft gemeinsam ist … eine Konstellation von Meinungen, Wertungen und Methoden…“ (Thomas Kuhn 1976)



  - Viele Programmiersprachen unterstützen Konstrukte aus unterschiedlichen Paradigmen.

universität wien

Das logikbasierte Paradigma (logic programming)

- ein Programm besteht aus Regeln und Fakten

Regeln
- Wenn es regnet, nehme ich den Schirm.
- Wenn ich zerstreut bin, vergesse ich den Schirm unterwegs.
- Wenn ich zerstreut bin, grüße ich Bekannte nicht.
- Wenn ich nicht zerstreut bin, grüße ich Bekannte.
- Wenn es regnet und ich meinen Schirm unterwegs vergesse, werde ich nass.
- Wenn es schwül ist, bin ich zerstreut.

Fakten
- Es ist schwül.
- Es regnet.

  - Anfrage: werde ich nass? – Antwort: ja.
  - Ableitbare Fakten: Ich nehme den Schirm. – Ich bin zerstreut. – Ich vergesse den Schirm unterwegs. – Ich werde nass. – Ich grüße Bekannte nicht.

- ein Problem: Widersprüche zwischen den Regeln
  - Beispiel (nicht ganz ernst zu nehmen, aber illustrativ):
    - Regel 1: Der Chef hat immer recht.
    - Regel 2: Stimmt dies ausnahmsweise nicht, so findet Regel 1 Anwendung.

Meta-Regel

universität wien

Das funktionale Paradigma (functional programming)

- Beispiel: Cäsar-Verschlüsselung

„Nimm jeweils den im Alphabet drittfolgenden Buchstaben“

```
CAESAR
↓↓↓↓↓↓
FDHVDU
```

jedes Vorkommen von 'A' kann durch 'D' ersetzt werden

- Das Programm ist nur aus Funktionen (Abbildungen) im mathematischen Sinn aufgebaut. Jedes Vorkommen eines Funktionsaufrufes kann durch das Funktionsergebnis ersetzt werden. Funktionen haben keine Seiteneffekte; es gibt keine Variablen.

- Beispiel: Fakultätsfunktion $n! = \begin{cases} n=0 \rightarrow 1 \\ n>0 \rightarrow n \cdot (n-1)! \end{cases}$

3! » 3·(3-1)! » 3·2! » 3·2·(2-1)! » 3·2·1! » 3·2·1·(1-1)! » 3·2·1·0! » 3·2·1·1 » 6

» hier: „kann ersetzt werden durch“

universität wien

Das imperative Paradigma (imperative programming)

- „Pfadfinder-Geländespiel“:
  - Geh zur alten Höhle
  - Merk dir, wieviele Fichten davor stehen
  - Geh den Weg weiter bis zur Gabelung
  - Zähle die Fichten, die du hier siehst, dazu
  - Wenn du insgesamt fünf Fichten gezählt hast, geh den linken Weg weiter; wenn sieben, dann den rechten
- Der Lösungsweg ist durch eine Folge von Anweisungen (bzw. Befehlen) vorgegeben
- Anweisungen können Werte in Variablen zwischenspeichern, lesen und verändern

universität wien

Summe der Zahlen von 1 bis n

```
int summe = 0;
while(n > 0) {
   summe += n;
   n--;
}
cout << summe;
```

C++

```
(reduce + 0 (iota n 1))
```

Scheme (Lisp Derivat)

```
sum(1,1).
sum(A,Result) :-
     A > 0,
     Ax is A - 1,
     sum(Ax,Bx),
     Result is A + Bx.
```

Prolog

universität wien

Programmiersprachen – Klassifikation

- Neben dem verwendeten Programmier-Paradigma gibt es noch weitere Klassifikationskriterien

- sequentiell versus parallel

- typisiert versus untypisiert (fließender Übergang)

- orthogonal dazu: objektorientiert

universität wien

Kategorisierung von C++

- *imperativ, prozedural* (Wertzuweisung an Variablen, beliebig viele Zuweisungen pro Variable)
- *sequentiell* (Anweisungen werden in der von der Programmiererin vorgegebenen Reihenfolge ausgeführt)
- *streng typisiert* (Variablen haben vordefinierte Wertebereiche, die zunächst eng mit den Operand-Typen der gängigen Prozessoren korrespondieren; byte, int, long int, float ...)

Im Folgenden nur mehr: imperatives Paradigma

universität wien

Structured program theorem

- Satz (Böhm/Jacopini 1966):
  - Ein beliebiger Programmablauf (vgl. flow-chart) lässt sich mit nur drei Programmkonstrukten verwirklichen:

  - Die **Sequenz** von Anweisungen ist eine Anweisung
  - Die **Entscheidung** ist eine Anweisung
  - Die **Iteration** ist eine Anweisung

universität wien

Wiederholung

• Algorithmus	Eindeutige, schrittweise Vorschrift
• (imperatives) Programm	Implementierung eines Algorithmus
• Programmiersprache	Formalisierte Sprache zur Erstellung von Programmen
• Compiler	Übersetzung von Sourcecode in Maschinencode
• Interpreter	„Direkte“ Ausführung von Sourcecode
• Sprachparadigmen	imperativ versus funktional versus logik-basiert
• Programmkonstrukte	Sequenz, Entscheidung, Iteration

universität
wien

2. „Hello World“

Ein erstes C++ Programm

universität wien

„Hello world“ (1)

```
#include<iostream>
using namespace std;

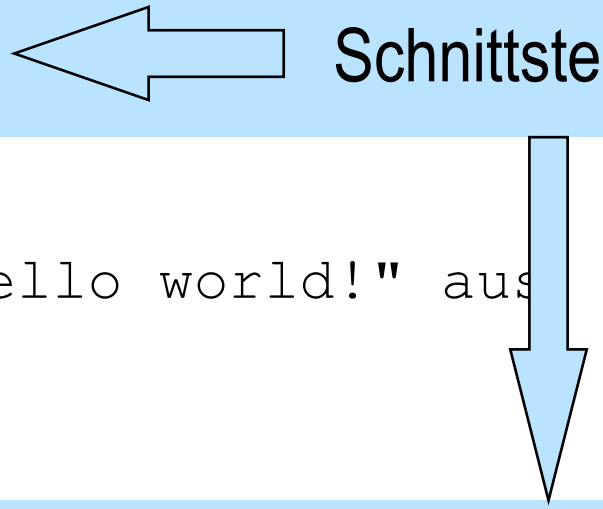
//Dieses Programm gibt "Hello world!" aus
int main()
{
    cout << "Hello world!\n";
    return 0;
}
```

universität wien

„Hello world“ (2)

```
#include<iostream>
using namespace std;

//Dieses Programm gibt "Hello world!" aus
int main()
{
    cout << "Hello world!\n";
    return 0;
}
```



universität wien

„Hello world“ (3)

```
#include<iostream>
using namespace std;

//Dieses Programm gibt "Hello world!" aus
int main()
{
    cout << "Hello world!\n";
    return 0;
}
```

Kommentar

Alles von // bis zum Ende der Zeile wird vom Compiler ignoriert (Alternativ /* … */ für mehrzeilige Kommentare)

„Hello world“ (4)

```
#include<iostream>
using namespace std;

//Dieses Programm gibt "Hello world!" aus
int main()
{
    cout << "Hello world!\n";
    return 0;
}
```

Funktion

main() wird beim Programmstart aufgerufen

Returnwert prinzipiell beliebig (Konventionsgemäß bedeutet 0 OK)
Aus historischen Gründen darf return-Statement in main auch fehlen

universität wien

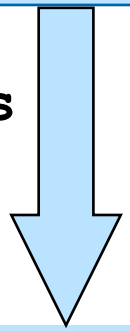
„Hello world“ (5)

```
#include<iostream>
using namespace std;

//Dieses Programm gibt "Hello world!" aus
int main()
{
    cout << "Hello world!\n";
    return 0;
}
```

Ausgabe

einer Zeichenkettenkonstante
(\n steht für **ein** Zeichen – newline. Es bewirkt einen Zeilenvorschub, wenn es ausgegeben wird.)



universität wien

Programmstart

- Um das Programm zu starten, muss es kompiliert (übersetzt) und gelinkt (gebunden) werden. Auf den in den Übungen verwendeten Rechnern geht das mit einem einzigen Befehl (Annahme: der Sourcecode befindet sich in einer Datei namens “hello_world.cpp”):

```
clang++ hello_world.cpp –o hello_world
```

- Das so erstellte Programm kann gestartet werden mit:

```
./hello_world
```

- Es gibt (nicht überraschend) “Hello world!” am Bildschirm aus.

universität wien

Konventionen für Dateinamen

- Vermeiden von Leer- und Sonderzeichen macht das Leben leichter!

- Linux unterscheidet zwischen Groß- und Kleinschreibung! Verwendung von Kleinschreibung, soweit nicht gute Gründe für Großbuchstaben vorliegen, ist empfohlen.

• hello_world.cpp	C++ Sourcecode (auch .cc, oder - auf Linux - .C gebräuchlich)
• hello_world.c	C Sourcecode
• hello_world.o	kompilierte, nicht gebundene Datei (object file)
• hello_world.h	Headerdatei (vgl iostream; auch .hh, .hpp gebräuchlich)
• hello_world	Exekutierbare Datei (executable)

universität wien

Programmfehler Kategorisierung

- compile time (beim Übersetzen)
  - **Warning:** Das Programm ist syntaktisch korrekt (wohlgeformt), aber es enthält Konstrukte die eventuell zu run time Fehlern führen. (z.B. = statt ==). Es wird ein lauffähiges Programm erzeugt.
  - **Error:** Der Text ist syntaktisch nicht korrekt (nicht wohlgeformt) und somit auch kein Programm. Es kann kein lauffähiges Programm erzeugt werden.
- link time (beim Binden)
  - **Warning:** Fehler, der eventuell für Überraschungen sorgt, aber nicht die Erzeugung eines lauffähigen Programms verhindert.
  - **Error:** Fehler, der keine Erzeugung eines lauffähigen Programms erlaubt
- run time (zur Laufzeit)
  - Fehler in der Programmlogik
  - “Bedienungs”fehler

universität wien

“Bedienungs”fehler: Programme umfassend absichern

- Keine leichte Aufgabe
- Abwandlungen von Murphy’s law:
  - "*Nothing is foolproof to a sufficiently talented fool.*“
  - "*If you make something idiot-proof, someone will just make a better idiot.*“

- “*A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.*“ (Douglas Adams in Mostly Harmless - The fifth book in the increasingly inaccurately named Hitchhikers Trilogy)

universität wien

Laufzeitfehler (run time errors)

- Sind in der Regel am schwierigsten zu finden.

- Fehler in der Programmlogik
  - Das Programm liefert nicht die gewünschten Ergebnisse. Der Programmablauf kann durch zusätzliche Ausgaben (Logging), Zustandsüberprüfungen (assert) oder die Verwendung von Debuggern getestet werden.

- „Bedienungs“fehler
  - Das Programm wird durch unvorhergesehene Ereignisse (z.B. falsche Eingaben, Ausfall der Internetverbindung) in einen unerwarteten bzw. undefinierten Zustand versetzt. Defensive Programmierung und geeignete Behandlung von Ausnahmezuständen helfen, Programme gegen diese Art von Fehlern robust zu machen.

- Ausgiebiges Testen erforderlich!
- Achtung: Testen kann niemals die Korrektheit eines Programms beweisen.

universität wien

Style matters

```
#include<iostream>
using namespace std;int main(){cout<<
"Hello world!\n";return 0;}
```

- Im Sinne der Lesbarkeit und der Wartbarkeit empfiehlt sich die Verwendung von Formatierungskonventionen.
- Genaue Konventionen können je nach verwendeter Literatur oder persönlichen Vorlieben unterschiedlich sein.
- Wichtig ist, die gewählte Konvention auch konsistent durchzuhalten.

universität wien

Wiederholung

• Libraries einbinden	#include <iostream>/using namespace std;
• Kommentare	zur Dokumentation /*…*/ //…
• Funktion	int main();
• main()	wird beim Programmstart aufgerufen
• Übersetzen und Linken	clang++ x.C –o x
• Programm starten	./x
• Namenskonventionen	x.cpp / x.h / x.o / x
• Fehlerkategorien	compile time / link time / run time
• Testen	kein Korrektheitsbeweis, aber unabdingbar
• Style matters	Lesbarkeit / Wartbarkeit

universität
wien

3. Datentypen, Variable und Ausdrücke

universität wien

Datentyp

- Ein Datentyp <W,O> ist eine Menge von Werten (Wertebereich W) mit einer Menge von Operationen (O), die auf diesen Werten definiert sind.

Beispiele: <R, {+,-,*,/}>

Ausprägungen: 3 + 4, 3.5 * 6 / 10, ...

<{wahr, falsch}, {∧, ∨, ¬}>

Ausprägungen: wahr ∨ falsch, wahr ∧ ¬ falsch,

...

- Werte nennen wir auch Exemplare oder Instanzen eines Datentyps.

universität wien

Datentypen und Speicherplatz

- Wieviel Speicherplatz eine Instanz eines bestimmten Datentyps benötigt, hängt vom Wertebereich des Datentyps ab
  - (exakt: von der Implementation des Datentyps in einer Programmiersprache)
- Die Größe des Speicherplatzes wird in Bytes angegeben
  - 1 Byte umfasst 8 Bits (nach C++ Standard mindestens 8 Bits)
  - Ein Bit kann den Wert 0 oder 1 annehmen
  - 1 Byte kann daher $2^8 = 256$ verschiedene Werte annehmen
- Beispiele
  - Größe(<{0..255},{+,-,*,/}>) = 1 Byte
  - Größe(<{-32768..32767},{+,-,*,/}>) = 2 Byte
  - Größe(<{wahr, falsch},{∧,∨,¬}>) = 1 Byte (warum?)

universität wien

Fundamentale Datentypen (fundamental data types)

- Auch als primitive (primitive) bzw. eingebaute (built-in) Datentypen bezeichnet

- Auswahl:

Typ	Größe	Wertebereich
char	1	mindestens 256 unterschiedliche Zeichen
int	4	-2,147,483,648 bis 2,147,483,647
double	8	$-1.7 \cdot 10^{308}$ bis $1.7 \cdot 10^{308}$
bool	1	true, false

- Größen (und damit Wertebereiche) implementationsabhängig (implementation defined)

universität wien

Datentyp Modifier

- Für den Datentyp `int` können die Modifier `short`, `long` und `long long` als Größenspezifikation verwendet werden. Bei Verwendung eines Modifiers darf das Schlüsselwort `int` entfallen (also z.B. nur `short` statt `short int`)

- Für `char` und `int` gibt es die Modifier `signed` und `unsigned`. Diese können bei `int` zusätzlich zu den anderen Modifiern verwendet werden. Das Schlüsselwort `int` darf wieder entfallen. Für `int` ist der Default `signed` für `char` ist der Default implementationsabhängig.

- Für `double` gibt es den Modifier `long`. Will man Speicherplatz sparen, kann man den Datentyp `float` verwenden.

universität wien

Benutzerdefinierte Datentypen (user defined)

- Komplexe Typen, die aus fundamentalen und/oder anderen benutzerdefinierten Datentypen “zusammengesetzt” werden.

- z.B.: const char[7]

- In der C++ Standard Library werden eine ganze Reihe von Datentypen definiert.

- z.B.: string, vector, complex

universität wien

Darstellung von Werten

- In (den üblichen) Computern werden Werte prinzipiell als Bitmuster dargestellt.
- Die Anzahl von Bits, die verwendet werden und deren Interpretation werden durch den Datentyp des Werts definiert.

0100000000000100000000000000000000000000000000000000000000000000

== 0x4004000000000000

== 4612811918334230528 (ganze Zahl) == 2.5 (Gleitkommazahl)

- Vgl. TGI Zahlendarstellungen

universität wien

Literale (literals)

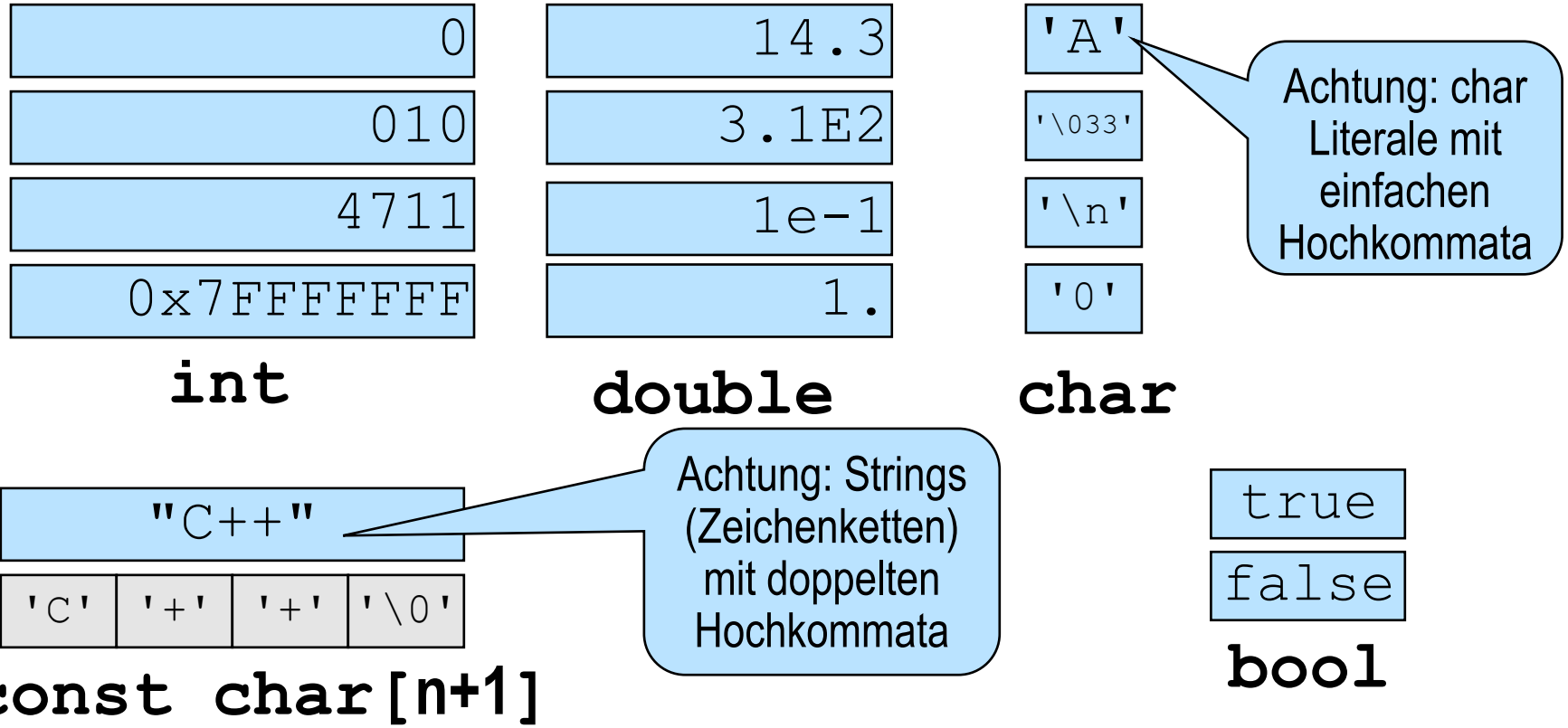
- Ein Literal ist ein konstanter Wert, der direkt im Programmtext auftritt.

- "Hello world!"
- 42
- 3.14

universität wien

Datentypen von Literalen

- Wie allen Werten werden auch Literalen Datentypen zugewiesen:



→ : Escape-Sequenzen

→ Datentyp Suffixe

universität wien

sizeof Operator

- Der Operator **`sizeof`** liefert die Größe eines Datentyps. Er kann direkt auf einen Datentyp oder einen Ausdruck (Wert) angewendet werden:

```
cout << "sizeof long double: " << sizeof(long double) << '\n';
short s;
cout << "sizeof s: " << sizeof(s) << '\n';
cout << "sizeof lf string: " << sizeof("\n") << "\n";
cout << "sizeof lf char: " << sizeof('\n') << '\n';
cout << "sizeof 2*3: " << sizeof(2*3) << '\n';
cout << "sizeof 2*3ul: " << sizeof(2*3ul) << '\n';
```

- Ausgabe:

```
sizeof long double: 16
sizeof s: 2
sizeof lf string: 2
sizeof lf char: 1
sizeof 2*3: 4
sizeof 2*3ul: 8
```

Größenangaben sind relativ zum Datentyp char, dessen Größe mit 1 definiert wird.

universität wien

Garantien zur Größe von Datentypen

- C++ macht nur wenige Garantien zur Größe von Datentypen

- Allgemein gilt nur:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
char mindestens 8 bit;
short, int mindestens 16 bit;
long mindestens 32 bit
long long mindestens 64 bit
```

- Ab C++11 gibt es auch Datentypen mit fix definierten Größen (z.B.: int16_t), nicht alle davon müssen aber auch auf jedem Rechner zur Verfügung stehen.

Anmerkung: Der C++ Standard definiert Byte als „*addressable unit of data storage large enough to hold any member of the basic character set of the execution environment*“ somit gilt immer sizeof(char) ist ein Byte. Byte kann aber unterschiedlich viele Bits enthalten, im Unterschied zum gängigen Gebrauch.

universität wien

Variable (variable)

- Eine Variable ist ein veränderbarer Wert.
- Die Variable hat einen Datentyp und kann einen Wert, sowie einen eindeutigen Namen (Bezeichner, identifier) haben.
- Zur Laufzeit hat jede Variable eine eindeutige Speicheradresse zugeordnet.

universität wien

Variablenvereinbarung (variable definition)

- Um dem Compiler zu erlauben, für benannte Variablen genau den benötigten Speicherplatz festzulegen und Werte richtig zu deuten, müssen diese vereinbart (definiert) werden.

- Variablenvereinbarung (einfachste Form):

  Datentyp *Variablenliste*`;`

- Beispiele:

```
double x;
int a, b, c;
short int s;
```

- Die Vereinbarung legt hier bereits den benötigten Speicherplatz an.

universität wien

Deklaration vs. Definition (declaration vs. definition)

- Eine Deklaration führt einen Namen in das Programm ein und teilt dem Compiler mit, wofür der Name steht (z.B. x bezeichnet eine Variable vom Typ double). Das Konstrukt, auf welches sich der Name bezieht, wird aber nicht erzeugt (z.B. für x wird kein Speicher alloziert, weil das in einem separat kompilierten Programmteil passiert).

```
extern double x; //x wird woanders definiert
```

- Eine Definition ist immer auch eine Deklaration, erzeugt aber gleichzeitig auch das durch den Namen referenzierte Konstrukt

```
double y {0};  //y wird mit dieser Zeile definiert
               //und initialisiert
```

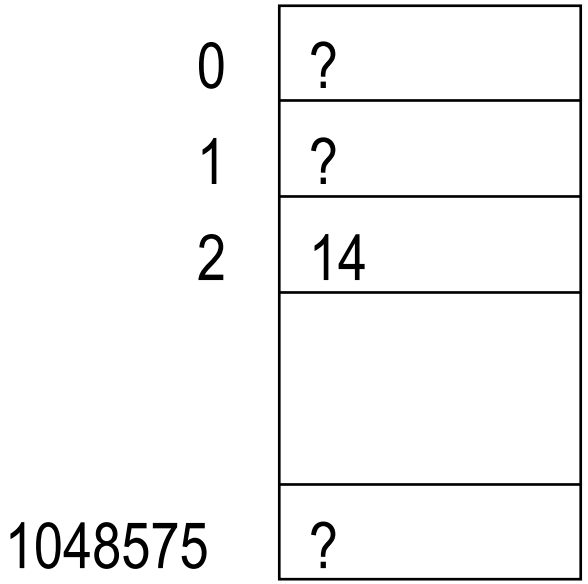
universität wien

Variablen im Speicher

- Falls die Variable benannt ist, kann sie durch ihren Namen angesprochen werden.
- Variablen können stets durch ihre Adresse im Hauptspeicher angesprochen werden.

- Beispiel:
Variable X für <{0..255},{+,-,*,/}>
addr(X) = 2, Inhalt(X) = 14, Größe(X) = 1

- Situation im Hauptspeicher:



universität wien

Initialisierung

- Wird eine Variable nicht initialisiert, so ist ihr Wert undefiniert (zufälliges Bitmuster). Versuche, den Wert zu verwenden (z.B. auszugeben oder Berechnungen auszuführen) resultieren in undefiniertem Verhalten

- Variablen sollten daher immer initialisiert werden

- Beispiel: `int x {3}, y {x * 7};` x: 3 y: 21

- so nicht: ~~`int x; int y {x * 7};`~~ x: ??? y: ???

- Ausnahme: Variablen die zum Einlesen von Werten dienen:

```
int x;
cin >> x; //x erhält einen Wert
```

→ Initialisierungssyntax

universität wien

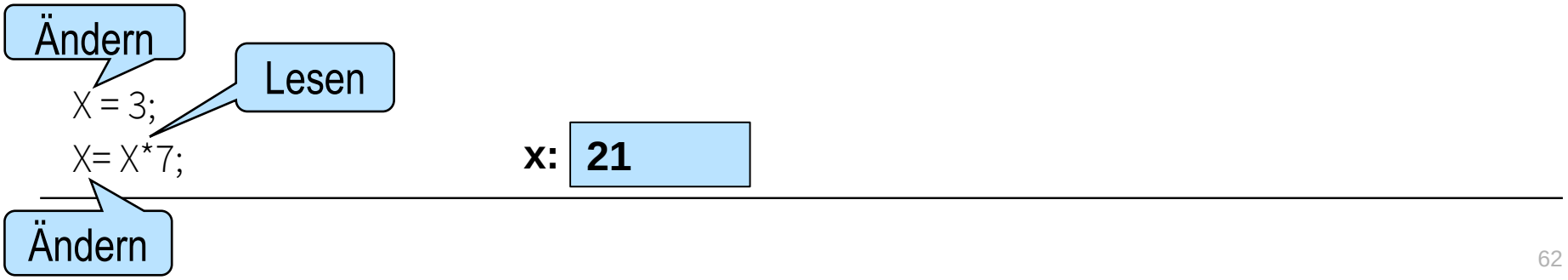
Wertzuweisung (assignment)

- Wir unterscheiden zunächst zwei Verwendungsarten für Variablen:

  Lesen einer Variable (Zugriff auf ihren aktuellen Inhalt)
  Wertzuweisung auf eine Variable (Ändern ihres Inhalts, Schreiben der Variable, Belegen der Variable mit einem Wert)

```
V    =  Ausdruck;
```
linke Seite — rechte Seite

- Vorgangsweise bei der Wertzuweisung:
  1. Berechne den Wert des Ausdrucks auf der rechten Seite
  2. Weise den berechneten Wert der Variablen auf der linken Seite zu



universität wien

Initialisierung versus Zuweisung

- Bei der Initialisierung wird eine neue Variable mit einem Anfangswert belegt.
- Bei der Zuweisung wird ein anderer Wert in die Variable eingetragen. Die Variable hat (so sie nicht uninitialisiert ist) zuvor schon einen Wert.
- Bei primitiven Datentypen gibt es da keinen großen Unterschied. Wenn die Variable aber ein Objekt enhält, dann muss das Objekt zuerst zerstört werden, bevor ein neues Objekt eingetragen werden kann (das wird uns später noch beschäftigen).
- Bei der uniform initialization ist der Unterschied syntaktisch klarer. In jedem Fall wird bei der Initialisierung immer ein Datentyp für die neu definierte Variable angegeben.

universität wien

Benannte Konstanten

• Benannte Konstanten werden mit dem Schlüsselwort **`constexpr`** vereinbart. Der Wert der Konstanten muss bei der Definition initialisiert werden und kann danach nicht mehr verändert werden.

```
constexpr char newline {'\n'};
constexpr double pi {3.1415};
```

• Ausdrücke, die nur aus (benannten oder unbenannten) Konstanten bestehen, heißen konstante Ausdrücke. Sie können vom Compiler ausgewertet werden.

```
constexpr double rezpi {1.0 / pi};
```

• Es ist auch möglich, Konstante erst zur Laufzeit festzulegen. Dazu wird das Schlüsselwort **`const`** verwendet.

```
double x;
cin >> x;
const double rezx {1.0 / x};
```

universität wien

Magic values vs. benannte Konstante

- Was ist schlimm an Werten, die ohnehin jedeR kennt?

- 3.14159265358979323846364, 12, -1, 365, 24, 2.7182818284590, 299792458, 86400, 4.669201, 2.54, 1.618, -273.15, 6.6260693e-34, 0.5291772108e-10, 6.0221415e23, 0.5772156649015328 6, 1.30637788386308069046861449 26

- Vielleicht kennt sie doch nicht jedeR
- Änderungen der Genauigkeitsanforderungen

- Benannte Konstanten mindern diese Probleme.

universität wien

Magic values vs. benannte Konstante

- Was ist schlimm an Werten, die ohnehin jedeR kennt?

- 3.14159265358979323846264, 12, -1, 365, 24, 2.7182818284590, 299792458, 86400, 4.669201, 2.54, 1.618, -273.15, 6.626070**15**e-34, 0.5291772108e-10, 6.0221415e23, 0.5772156649015328**6**, 1.3063778838630806904686144926

- **Sicher, dass keine der obigen Konstanten einen Tippfehler enthält oder sich verändert hat?**
- Vielleicht kennt sie doch nicht jedeR
- Änderungen der Genauigkeitsanforderungen

- Benannte Konstanten mindern diese Probleme.

universität wien

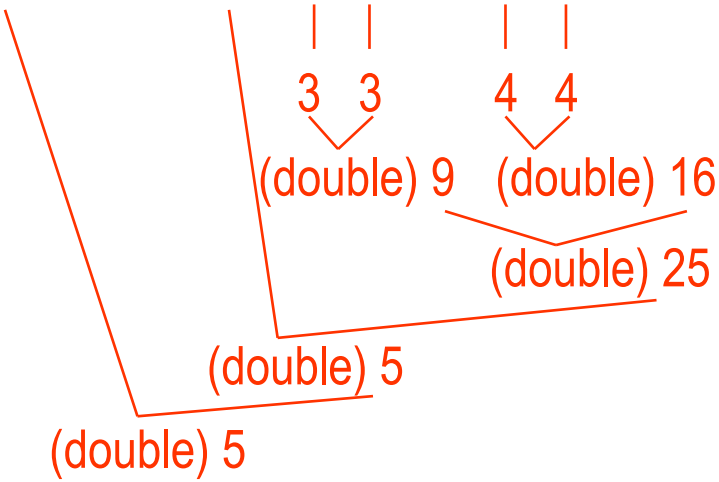
Ausdrücke (expressions)

• Ausdrücke sind Verknüpfungen von Operanden durch entsprechende Operatoren. Ihre Auswertung liefert einen (typisierten) Wert. Operanden sind (evt. geklammerte) Teilausdrücke, Variablen, Konstanten und Funktionsaufrufe.

Beispiel:

```
double a {3}, b {4}, c;
c = sqrt( a*a + b*b );
```

Auswertung:



universität wien

Operatoren

- Operatoren verknüpfen Operanden mit passenden Datentypen und berechnen einen Wert (der ebenfalls wieder einen passenden Datentyp hat). Manche Operatoren (z.B. =, ++, Ausgabe) bewirken auch Seiteneffekte.
- Die Bedeutung von Operatoren kann je nach Datentyp unterschiedlich sein (überladen, overloading), z.B.:

```
double a {3}, b {4};
cout << a+b; //7 Addition
cout << a-b; //-1 Subtraktion
```

Zeichenkettenliterale haben **nicht** den Datentyp string, sondern const char[n+1]. Bei Bedarf kann aber automatisch aus const char[n+1] ein string erzeugt werden (nicht umgekehrt).

```
string s1 {"ab"}, s2 {"er"};
cout << s1+s2;  //"aber" Verkettung (concatenation)
// cout << s1-s2; //nicht erlaubt
//cout << "ab"+"er"; //nicht erlaubt
cout << "ab"-"er"; //erlaubt (Interpretation siehe später)
```

universität wien

Übersicht der meistverwendeten Operatoren (1)

Operator	Name	Kommentar
`f(a)`	Funktionsaufruf (function call)	Funktion wird mit Parameterwert a aufgerufen
`++lval`	Prefix-Inkrement (pre-increment)	Um eins erhöhen und den erhöhten Wert verwenden
`--lval`	Prefix-Dekrement (pre-decrement)	Um eins vermindern und den verminderten Wert verwenden
`lval++`	Postfix-Inkrement (post-increment)	Um eins erhöhen und den ursprünglichen Wert verwenden
`lval--`	Postfix-Dekrement(post-decrement)	Um eins vermindern und den ursprünglichen Wert verwenden
`!a`	Nicht (not)	Logisches Nicht (liefert einen bool Wert)
`-a`	Unäres Minus, Vorzeichen (unary minus, sign)	
`a*b`	Multiplikation (multiplication)	
`a/b`	Division (division)	Achtung: bei int Operanden werden Nachkommastellen abgeschnitten
`a%b`	Modulo (modulo)	Rest (remainder) der ganzzahligen Division, nur für ganzzahlige Typen definiert
`a+b`	Addition	Bei string: Verkettung
`a-b`	Subtraktion	

universität wien

Übersicht der meistverwendeten Operatoren (2)

Operator	Name	Kommentar
`out<<b`	b ausgeben (output b)	Wenn „out“ den Datentyp ostream hat, Ergebnis ist out
`in>>b`	b einlesen (input b)	Wenn „in“ den Datentyp istream hat, Ergebnis ist in
`a<b`	kleiner (less than)	Liefert einen bool Wert
`a<=b`	kleiner gleich (less than or equal)	Liefert einen bool Wert
`a>b`	größer (greater than)	Liefert einen bool Wert
`a>=b`	größer gleich (greater than or equal)	Liefert einen bool Wert
`a==b`	gleich (equal)	Liefert einen bool Wert (nicht mit Zuweisung = verwechseln!)
`a!=b`	ungleich (not equal)	Liefert einen bool Wert
`a&&b`	logisches und (logical and)	Liefert einen bool Wert, Parameter sind bool
`a||b`	logisches oder (logical or)	Liefert einen bool Wert, Parameter sind bool
`lval=a`	Zuweisung (assignment)	lval erhält den Wert von a, der Wert des Ausdrucks ist der neue Wert von lval
`lval*=a`	Zusamengesetzte Zuweisung (compound assignment)	„Abkürzung“ für lval=lval*a, auch für %,/,+,-,…

universität wien

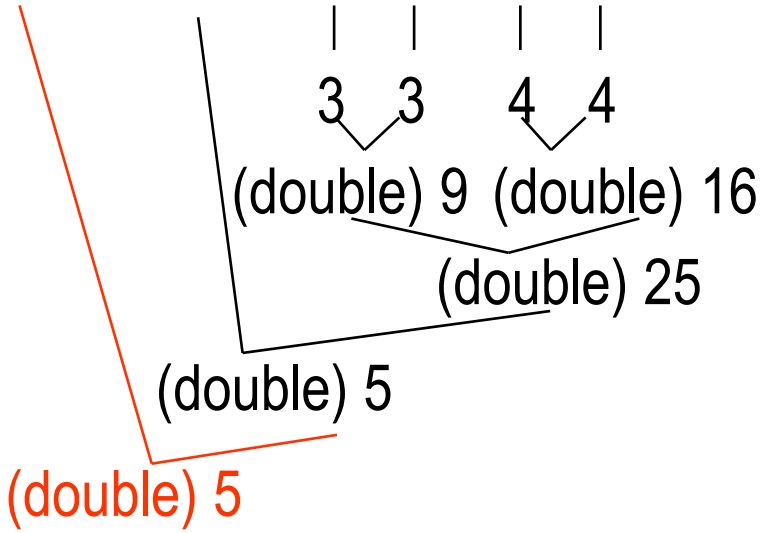
Zuweisungsoperator

• In C++ sind Zuweisungen auch Ausdrücke und liefern den Wert der linken Seite der Zuweisung nach erfolgter Wertübertragung. Die Wertübertragung gilt als Seiteneffekt des Zuweisungsoperators =.

Beispiel:

```
double a {3}, b {4}, c;
c = sqrt( a*a + b*b );
```

Auswertung:



universität wien

Ausdruckauswertung (1)

- Die Auflösung von ungeklammerten Ausdrücken in Teilausdrücke wird durch die Bindungsstärke und links/rechts-Assoziativität der Operatoren bestimmt.
- In erster Linie regelt die Bindungsstärke (Operatorpriorität) die implizite Klammerung von Teilausdrücken.

```
3 + 4 * 2          entspricht   3 + (4 * 2)
x * y + 5 * z      entspricht   (x * y) + (5 * z)
```

- Bei Operatoren gleicher Bindungsstärke regelt die links/rechts-Assoziativität die implizite Klammerung von Teilausdrücken.

```
3 - 4 - 2  entspricht  (3 - 4) - 2  linksassoziativ
x = y = z  entspricht  x = (y = z)  rechtsassoziativ
```

- Tipp: Klammern setzen, wenn Priorität oder Assoziativität unklar

→ Operatorenübersicht

universität wien

Ausdruckauswertung (2)

- Die Auswertungsreihenfolge von Teilausdrücken ist in C++ im Allgemeinen jedoch undefiniert.
- Konstruktionen, deren Ergebnisse von der Auswertungsreihenfolge abhängen, liefern undefiniertes Verhalten.

- Negativbeispiel:

```
int x, y, z;
(x * y) + (5 * z)        // welche Multiplikation zuerst?
x = (y = 3) + (y = 4);   // hat y den Wert 3 oder 4?
```

Durch die Seiteneffekte wird die Auswertungsreihenfolge entscheidend

universität wien

Ausdrücke mit undefiniertem Verhalten

- Die genauen Regeln sind komplex und haben sich im Laufe der Zeit verändert, sodass z.B.:

```
i=i++ //definiert seit C++17
i=++i //definiert seit C++11
```

- Mit dem Standard C++17 wurden viele weitere Fälle wohldefiniert.

- Faustregel:

  Ausdrücke vermeiden, die

  1. einen Seiteneffekt auf eine Variable bewirken,
  2. die Variable noch einmal enthalten,
  3. die Reihenfolge der Abarbeitung nicht definieren

- (Ausnahme: Auf der rechten Seite einer Zuweisung darf dieselbe Variable noch einmal lesend verwendet werden: `i=i+7 //OK`)

universität wien

Implizite Typkonversionen (implicit type conversions)

Betrachten wir einen arithmetischen Ausdruck wie **`x + y`**,
wobei **`x`** und **`y`** zu verschiedenen arithmetischen Typen gehören.
Vor der Auswertung des Ausdrucks werden geeignete Typ-Konvertierungen durchgeführt (sogenannte implizite Konversionen, Standardkonversionen).

- Automatische Konversionen basieren auf einer Reihe von eher komplizierten „Konversionsregeln“. Für arithmetische Ausdrücke wie **`x + y`** gilt jedoch eine einfache Faustregel:

Ausweiten auf den ‚größeren‘ der beiden Datentypen

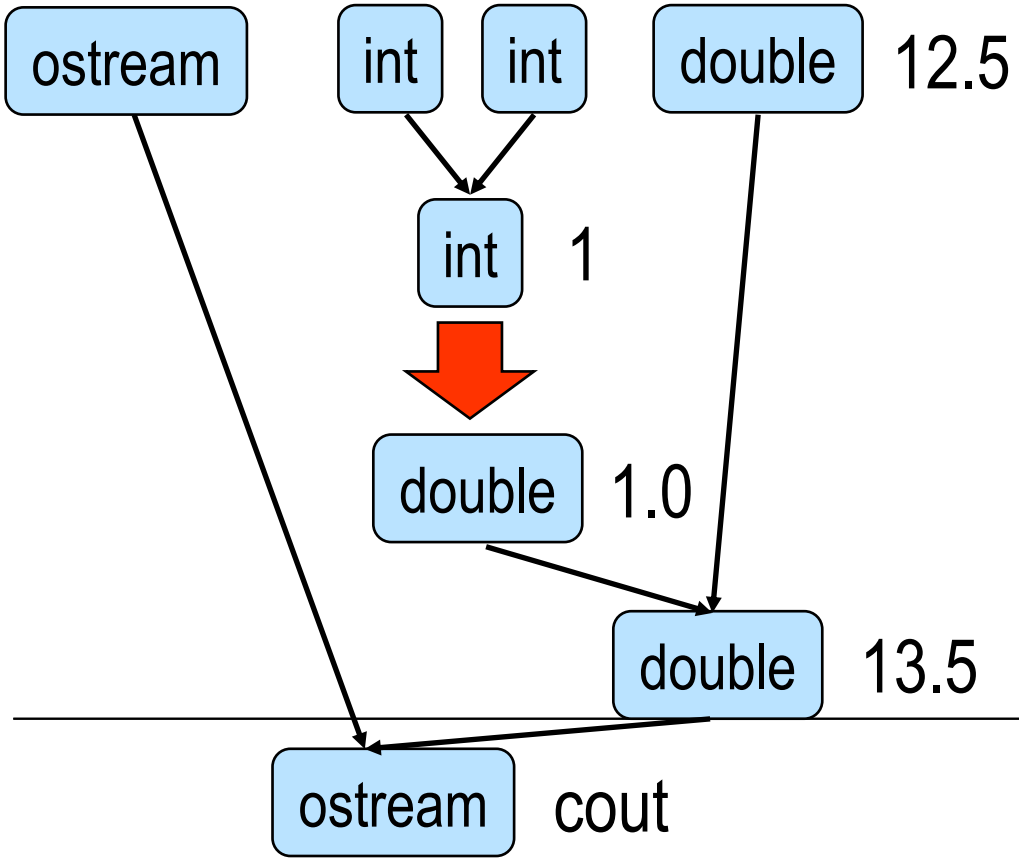
- Bei Zuweisungen und Initialisierungen gilt:

Konversion zum Datentyp der zu setzenden Variable/Konstante

universität wien

Typkonversionen – Variante 1

```
double x {12.5};
cout <<  3 / 2  + x;
```



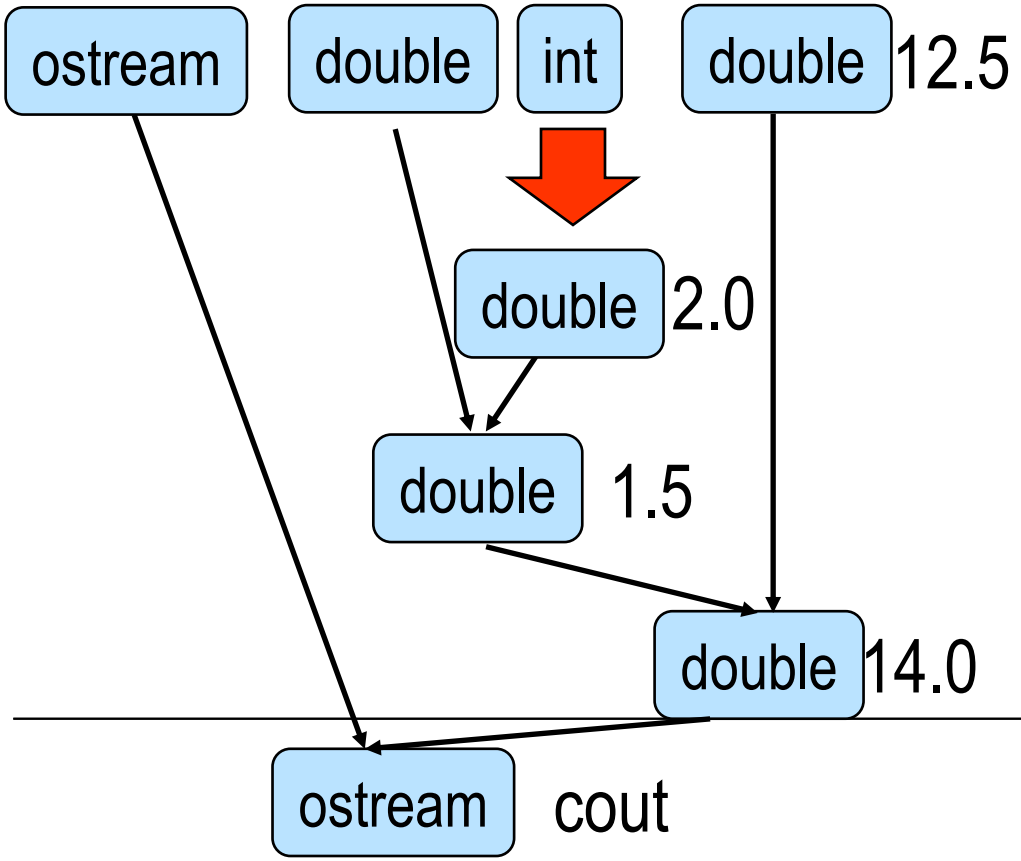
Die Division zweier ganzer Zahlen liefert wieder eine ganze Zahl, d.h. 3 / 2 == 1

Ausgabe: `13.5`

universität wien

Typkonversionen – Variante 2

```
double x {12.5};
cout <<  3. / 2  + x;
```



Ausgabe: **14**

universität wien

Typsicherheit (type safety)

- Es ist sicherzustellen, dass Werte nur in einer Art und Weise verarbeitet werden, wie es ihrem Typ entspricht, andernfalls können unliebsame Effekte auftreten.

- Statische Typsicherheit (static): Kontrolle der Typsicherheit durch den Compiler; für viele Anwendungen zu restriktiv.

- Dynamische Typsicherheit (dynamic): Kontrolle der Typsicherheit zur Laufzeit durch vom Compiler automatisch generierten Zusatzcode; in C++ aus Effizienzgründen nicht vollständig umgesetzt.

universität wien

Sichere und unsichere Konversionen

- Erweiternde (widening) Konversionen sind sicher (bei der Rückkonversion wird der ursprüngliche Wert wiederhergestellt):

  bool nach char, bool nach int, bool nach double

  char nach int, char nach double,

  int nach double (eventuell Genauigkeitsverlust für große Werte)

- Einengende (narrowing) Konversionen können unsicher sein (viele Compiler warnen):

  double nach int, double nach char, double nach bool

  int nach char, int nach bool

  char nach bool

```
int i {1.7} //verboten
int i = 1.7 //OK (abschneiden)
char i {1000} //verboten
char i = 1000 //OK Zeichen?
char i {27} //OK Zeichen Nummer 27
```

universität wien

Operatoren für explizite Typumwandlung (explicit cast)

- Implizite Typkonversionen bringen eine schreibtechnische Vereinfachung, aber die Gefahr einer unbemerkten Konversion und eines damit verbundenen, eventuell unbeabsichtigten Informationsverlustes mit sich.
- Explizite Konversionsangaben sind nicht nur oft notwendig,
man kann mit ihnen manche implizite Umwandlungen auch dokumentieren:

```
int i; unsigned c;
...
i = static_cast<int>(c); // redundant, aber für LeserIn informativ
```

- Weitere Operatoren const_cast, dynamic_cast, reinterpret_cast werden wir vorerst nicht verwenden.
- Frühere Schreibweise: (typ)ausdruck oder typ(ausdruck) im Wesentlichen eine Kombination aller Operatoren

universität wien

Automatische Typbestimmung

- Das Schlüsselwort auto erlaubt, bereits bekannte andere Typen zu übernehmen:

```
auto i {7}; //i ist ein int
auto j {5ul}; //j ist ein unsigned long
auto x {3.5}; //x ist ein double
auto y {'c'}; //y ist ein char
auto s {"abc"};  //s ist ein const char* nicht string!
```

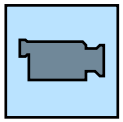
- Werden wir später verwenden.
- (Diese Form funktioniert erst ab C++17, wurde aber als Spezialfall rückwirkend auch in ältere Versionen von C++ eingeführt. Es darf kein = vor der geschwungenen Klammer verwendet werden und in den Klammern darf nur ein Element enthalten sein, weil die Bedeutung sonst eine andere ist. Will man ganz sicher gehen, empfiehlt sich die ältere Form der Initialisierung ohne geschwungene Klammern, also mit = oder mit runden Klammern)

universität wien

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```



universität wien

Beispiel: Datentypen (2)

Ausgabe:

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
0 ist gar nichts
0.5 ist auch nicht viel, aber immerhin
1 ist auch ohne Komma eine reelle Zahl
i ist gleich 1 und ist ungerade
Viel Spass mit C++
```

universität wien

Wiederholung

- Datentyp `int, double, ...`
- Literal `"Hello world", 42, 3.14`
- Variablenvereinbarung `int x, y; double z;`
- Initialisierung `int x {y};`
- Wertzuweisung `y = 3;`
- Konstante `constexpr int a {2};`
- Implizite Typkonversion `int y; y=7.4;`
- Explizite Typkonversion `static_cast`
- Ausdrücke `x + 3; y = y * 4;`
- Seiteneffekt `x = y = y * 4; ++x;`
- Automatische Typisierung `auto x = 8;`

universität
wien

4. Kontrollstrukturen

universität wien

Anweisungen (Statements)

- Ein C++ Programm besteht aus einer Reihe von Anweisungen, die sequentiell ausgeführt werden.

statement	=	expression-statement | compound-statement | `...`
statement-seq	=	statement_seq statement | statement
expression-statement	=	[expression] `';'`
compound-statement	=	`'{'` [statement_seq] `'}'`

- Gültige Statements daher:

```
;  //Leeranweisung
3; //Kein Effekt (Compilerwarnung)
{} //OK
```

→EBNF

universität wien

Kategorisierung von Statements

- Ausdruck mit Strichpunkt am Ende:
  - berechnet Werte, ändert Variable, Ein-/Ausgabe

- Deklarationen:
  - z. B.: Variablendefinition

- Kontrollstatements:
  - steuern den Ablauf des Programms

universität wien

Block / Verbundanweisung (compound statement)

- Fasst eine Sequenz von beliebig vielen Statments zu einem logischen Statement zusammen.

```
statement-seq    =  statement_seq statement |  statement
compound-statement   =      '{'  [statement_seq] '}'
```

- Kennen wir vom Block der Funktion main, kann aber überall eingesetzt werden, wo ein Statement benötigt wird.

universität wien

Auswahl (selection) – if-Anweisung

selection-statement = **'if'** **'('** condition **')'** statement [**'else'** statement]

```
if(2<3) i = 0; //OK
if(1) ; else j = 0; //OK
if(1) else j = 0; //Error
```

condition ist ein logischer Ausdruck, liefert also true oder false. Falls true geliefert wird, wird das erste Statement ausgeführt, sonst keines (falls der else-Zweig leer ist) oder das zweite.

universität wien

if-Anweisung Verschachtelung

selection-statement = **'if'** **'('** condition **')'** statement [**'else'** statement]

```
if (B1)
   if (B2)
      A1;
   else
      // B1 && !B2
      A2;
```

- Einrückung entspricht hier der logischen Bindung, ist aber irrelevant. Falls andere Bindung gewünscht ist:

```
if (B1)
   if (B2)
      A1;
   else ;
else
   // !B1
   A2;
```

```
if (B1) {
   if (B2)
      A1;

} else
   // !B1
   A2;
```

universität wien

Auswahl – switch Anweisung - Mehrwegverzweigung

selection-statement = 'switch' '(' condition ')' statement

- In der Regel liefert condition einen ganzzahligen Wert und statement ist ein Block mit verschiedenen Sprungzielen (Labels):

```
int n;
cin >> n;
switch (n) {
     case 0:  //fallthrough
     case 1: cout << "small"; break;
     case 7: cout << "medium"; break;
     case 12: cout << "large"; break;
     default: cout << "unknown";
}
```

- condition muss ganzzahlig, char oder Enumeration (siehe später) sein.
- Die case-Labels müssen konstante Ausdrücke und verschieden sein.
- break verhindert fallthrough.
- default ist optional (darf höchstens einmal vorhanden sein).

universität wien

Wiederholungsanweisungen / Schleifen (loops)

- iteration-statement = `'while'` `'('` condition `')'` statement |
  `'do'` statement `'while'` `'('` expression `')'` `';'` |
  `'for'` `'('` for-init-statement [condition] `';'` [expression] `')'` statement

- Prinzipiell wird nur ein Schleifenkonstrukt benötigt. Jede Form lässt sich in die anderen umformen. Je nach genauer Aufgabenstellung kann man die jeweils passende, konzise Formulierung wählen.

universität wien

while

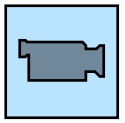
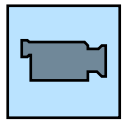
'while' '(' condition ')' statement

```
#include<iostream>
using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

Test: `n` ist `0`

Ergebnis (0) ist korrekt.

Eine **while**-Schleife kann auch **gar nicht** durchlaufen werden.

universität wien

do while

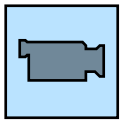
```
'do' statement 'while' '(' expression ')'';'
```

```
#include<iostream>
using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

Test: `n` ist `0`

Ergebnis (1) ist sehr ungenau.

Eine **do**-Schleife wird **immer mindestens einmal** durchlaufen.



for

```
'for' '(' for-init-statement [condition] ';' [expression] ')'
    statement
```

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

Initialisierung

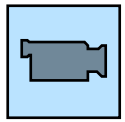
Test

Re-Initialisierung

Jeder der drei Teile in den Klammern darf auch leer bleiben. Die beiden Strichpunkte sind aber unbedingt erforderlich.
Ein leerer Test wird als true bewertet.

```
for (;;); //Endlosschleife
```

- Vgl. Mathematik: Vollständige Induktion
  (Initialisierung ~ Startschritt, Reinitialisierung ~ Induktionsschritt)
- Schleife ist allerdings (hoffentlich) endlich



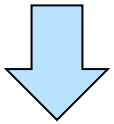
universität wien

break (1)

- `'break'';'`

- Manchmal ist es notwendig, aus einer Schleife an einer bestimmten Stelle auszubrechen und das Ändern der Abbruchbedingung und Überspringen von Statements, die nicht mehr ausgeführt werden sollen, führt zu schwerer lesbarem Code.

```
while (bedingung) {
   ...
   //hier soll beendet werden, falls x==0 ist
   ...
}
```



```
while (bedingung && x!=0) {
  ...
  if (x!=0) {
    ...
  }
}
```

universität wien

break (2)

- Das break-Statement kann hier Abhilfe schaffen. Es beendet die Schleife und setzt mit dem nächsten Statement nach der Schleife fort.

```
while (bedingung) {
    ...
    if (x==0) break;
    ...
}
```

- In C++ lässt sich break (leider) nur zum Beenden der jeweils innersten Struktur (Schleife bzw. switch-Anweisung) verwenden.

- break in Schleifen nur verwenden, wenn die Lesbarkeit dadurch gesteigert wird.

universität wien

continue

- `'continue'';'`

- Das continue-Statement startet den nächsten Schleifendurchlauf
- Die Statements nach dem continue-Statement bis zum Ende der Schleife werden übersprungen, wenn das continue-Statement ausgeführt wird. Die Schleife wird aber nicht beendet.

```
while (bedingung) {
    ...
    if (x==0) continue;
    ...
}
```

universität wien

logische Ausdrücke (1)

- Sowohl if, als auch die Schleifenkonstrukte verwenden logische Ausdrücke, deren Wert den Ablauf steuert.
- Ein logischer Ausdruck ist ein Ausdruck, der als boolescher Wert (true, false) interpretiert werden kann.
- Vergleichsoperatoren <, <=, >, >=, ==, != liefern boolesche Werte
- cin und cout können implizit nach bool umgewandelt werden (true, wenn keine Fehler aufgetreten sind, false sonst; "Einlesen" von EOF gilt als Fehler)
- double, int, char und Derivate können implizit nach bool umgewandelt werden (unsafe, narrowing conversion); 0 ergibt false, alles andere true (umgekehrt gilt: false wird zu 0 und true wird zu 1)

```
while (i) /*...*/;  //Idiom statt while (i!=0) /*...*/;
```

universität wien

logische Ausdrücke (2)

- Logische Operatoren !, && und || verknüpfen boolesche Operanden und liefern einen booleschen Wert.

	!
false	true
true	false

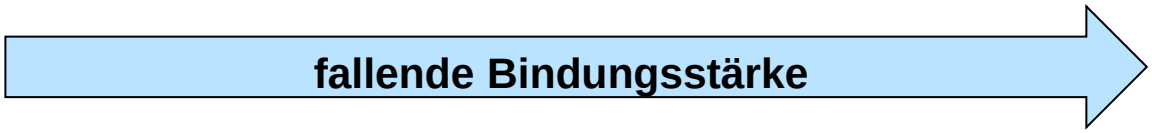
NICHT

&&	false	true
false	false	false
true	false	true

UND

||	false	true
false	false	true
true	true	true

ODER



universität wien

Logische Operatoren

- Für die binären logischen Operatoren && und || gelten spezielle Regeln:

- Während in C++ die Auswertungsreihenfolge der Operanden eines Operators im Allgemeinen nicht spezifiziert ist, wird bei && und || der linke Operand zuerst bewertet. Alle Seiteneffekte des linken Operanden werden abgeschlossen. Danach wird der rechte Operand nur dann bewertet, wenn das Ergebnis noch nicht feststeht.

```
true  || ??? immer true
false && ??? immer false
```

```
(i=0) || ++i //trotz zweier Seiteneffekte auf i wohldefiniert i==1
(i=0) && ++i //i==0
(i=1) || i++ //i==1
(i=1) && i++ //i==2
(i=0) || ++i && --i
```

universität wien

Geltungsbereich (scope)

- Der Geltungsbereich eines Namens definiert jenen Teil des Programmtextes, in dem der Name verwendet werden darf.

- Als Verwendung gilt dabei jede Aufnahme des Namens im Programmtext mit Ausnahme der Deklaration selbst.

- Unter anderem unterscheidet man in C++ zwischen
  - lokalem Geltungsbereich und
  - globalem Geltungsbereich

universität wien

Lokaler Geltungsbereich

- Ein Name hat lokalen Geltungsbereich, wenn er innerhalb eines Blocks vereinbart ist. Der Geltungsbereich beginnt dann mit der Vereinbarung des Namens und endet mit dem Block.
- Wird derselbe Name in einem inneren Block für ein anderes Objekt benützt, überlagert diese Vereinbarung die äußere (shadowing).

```
int i {1};                  // später nicht benutzt
int main() {
  int i {2};                    // überlagert äußeres int i
  {
    double i {3.1415};          // überlagert wiederum int i
    cout << i;
  }                             // Ende der Gültigkeit von double
  cout << i << "\n";
}                               // Ende der Gültigkeit inneres int i
```

Ausgabe: 3.14152

universität wien

Globaler Geltungsbereich

- Ein Name hat globalen Geltungsbereich, wenn er außerhalb sämtlicher Blöcke vereinbart ist. Der Geltungsbereich beginnt mit der Vereinbarung und endet mit dem Programmtext.
- Auf überlagerte globale Namen kann man mit dem Bereichsoperator `::` zugreifen.

```
int i {1};                  // wird unten benutzt
int main() {
  int i {2};                // überlagert äußeres int i
  {
    double i {3.1415};      // wird nicht benutzt
    cout << ::i;            // bezeichnet äußerstes i
  }                         // Ende der Gültigkeit von double i
  cout << i << "\n";
}                           // Ende der Gültigkeit inneres int i
```

Ausgabe: 12

universität wien

Anmerkung zum Geltungsbereich

- Der C++ Standard definiert eine Reihe von Geltungsbereichen, aus denen sich die für unsere Zwecke völlig ausreichende Unterscheidung von lokalem und globalem Geltungsbereich nur implizit ergibt.

  - block scope
  - function prototype scope
  - function scope
  - namespace scope
  - class scope
  - enumeration scope
  - template parameter scope

universität wien

Funktionen (1)

- Teile eines Programms können auch an anderer Stelle als Unterprogramm formuliert sein, wobei der Ablauf bei Bedarf dorthin verzweigt (Aufruf) und nach Abarbeitung des Unterprogramms wieder an die Stelle des Aufrufs zurückkehrt (Rücksprung).

- Arten von Unterprogrammen:
  - Einfache Prozeduren
  - Unterprogramme, die beim Rücksprung einen Wert an das aufrufende Programm zurückliefern (Funktionen)

- In C++ ist diese Unterscheidung unscharf, man spricht üblicherweise nur von Funktionen. C++ Funktionen, die keinen Wert zurückliefern, werden mit dem Datentyp `void` gekennzeichnet.

universität wien

Funktionsdefinition (1)

Ergebnistyp FName `'('` [Parameterdef { `','` Parameterdef }] `')'`
compound-statement

- vereinfachte Syntax
- muss i.a. vor (im Quelltext “oberhalb”) dem Aufruf stehen
- Ergebnistyp kann auch `void` sein (“Prozedur”)
- Parameterdefinition entspricht (lokaler) Variablenvereinbarung

universität wien

Funktionsdefinition (2)

```
double sqr(double a) {
  double b {a*a};
  return b;
}
```

- formale Parameter werden beim Aufruf durch aktuelle Parameter initialisiert
- **`return`** beendet die Funktion
  - Ist der Ergebnistyp nicht **`void`**, muss das Ergebnis durch **`return`** Ausdruck **`;`** zurückgegeben werden.
    - Erreicht die Funktion die abschließende geschwungene Klammer, ist das Verhalten undefiniert (Meist macht eine Compilerwarnung auf das fehlende **`return`**-Statement aufmerksam). Aus historischen Gründen gibt es eine Ausnahme für die Funktion **`main`**, die bei Erreichen der abschließenden geschwungenen Klammer 0 zurück liefert.
  - Ist der Ergebnistyp **`void`** (Prozedur), dann verlässt **`return;`** (bzw. das Erreichen der abschließenden geschwungenen Klammer) die Funktion ohne Rückgabe eines Ergebnisses

universität wien

Ablaufsteuerung

- Funktionsaufruf

  FName **'('** [Argument {**','** Argument }] **')'**

  – verzweigt zum Unterprogramm *FName*

  – initialisiert die formalen Parameter von *FName* mit den aktuellen Parametern (*Argument*).

    Dabei finden die üblichen impliziten Typumwandlungen (wie bei einer Zuweisung) statt.
    Narrowing conversions führen eventuell zu Compilerwarnungen

  – setzt nach Beendigung von *FName* das Programm unmittelbar hinter der Stelle des Aufrufs fort

  – kann einen Wert zurückliefern, der den Funktionsaufruf (im Kontext eines Ausdrucks) ersetzt

```
...
cout << sqrt(a*a+b*b);
...
```

universität wien

Funktionen (2)

```
#include<iostream>

using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}

int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

Programmablauf beginnt hier

Rückgabe

Ausgabe:
4 2 1 3 5

universität wien

Funktionen (3)

```
#include<iostream>
#include <cmath>

using namespace std;

double sqr(double a) {
  double b {a*a};
  return b;
}

int main() {
  cout << "A, B: ";
  double a, b;
  cin >> a >> b;
  cout << sqrt(sqr(a)+sqr(b));
  return 0;
}
```

`sqr()` stellt wie `main()` eine Funktion dar. Ihr Ergebnistyp ist `double`, der Ergebnistyp von `main()` ist `int`.

Die Variablen `a` und `b` in `sqr()` sind von den Variablen `a` und `b` in `main()` total unabhängig, alle vier sind jeweils nur innerhalb ihres Blocks bekannt.

universität wien

Funktionen (4)

```
#include<iostream>
#include <cmath>

using namespace std;

double sqr(double a) {
  double b {a*a};
  return b;
}

int main() {
  cout << "A, B: ";
  double a, b;
  cin >> a >> b;
  cout << sqrt(sqr(a)+sqr(b));
  return 0;
}
```

`sqr()` ist parametrisiert.

Ein **Parameter** ist ein Vehikel zur Kommunikation mit Unter-programmen. Im Funktionskopf werden **formale Parameter** definiert, die innerhalb der Funktion wie lokale Variablen wirken, jedoch beim Aufruf der Funktion durch je einen **aktuellen Parameter** initialisiert werden. (Für eventuell notwendige implizite Typumwandlungen gelten die Regeln für Zuweisungen)

Funktionen (5)

```
#include<iostream>
#include <cmath>

using namespace std;

int main() {
  cout << "A, B: ";
  double a, b;
  cin >> a >> b;
  cout << sqrt(sqr(a+b));
  return 0;
}

double sqr(double a) {
  return a*a;
}
```

Hier meldet uns der Compiler einen Fehler: Beim Versuch, **`main`** zu kompilieren, stößt er auf den Funktionsaufruf **`sqr`** – diese Funktion kennt er aber noch nicht, sie wird ja erst weiter unten definiert.

Lösung: entweder **`sqr`** vor **`main`** setzen oder …

universität wien

Funktionsdeklaration

```
#include<iostream>
#include <cmath>

using namespace std;

double sqr(double a);

int main() {
  cout << "A, B: ";
  double a, b;
  cin >> a >> b;
  cout << sqrt(sqr(a)+sqr(b));
  return 0;
}

double sqr(double a) {
  return a*a;
}
```

… wir **deklarieren** die Funktion `sqr` (und **definieren** sie später).

Eine **Definition** stellt ein Konstrukt (Variable, Funktion, etc.) tatsächlich zur Verfügung, während eine **Deklaration** den Übersetzer lediglich über diverse Eigenschaften eines (anderswo definierten) Konstruktes informiert.

Eine **Funktionsdeklaration** (ein **Prototyp**) entsteht aus einer Funktionsdefinition, indem der Funktionsrumpf durch `;` ersetzt wird.

Die Namen der formalen Parameter dürfen auch entfallen: `double sqr(double);`

Die Kombination der Parametertypen (`double`) einer Funktion wird **Signatur** der Funktion genannt.

universität wien

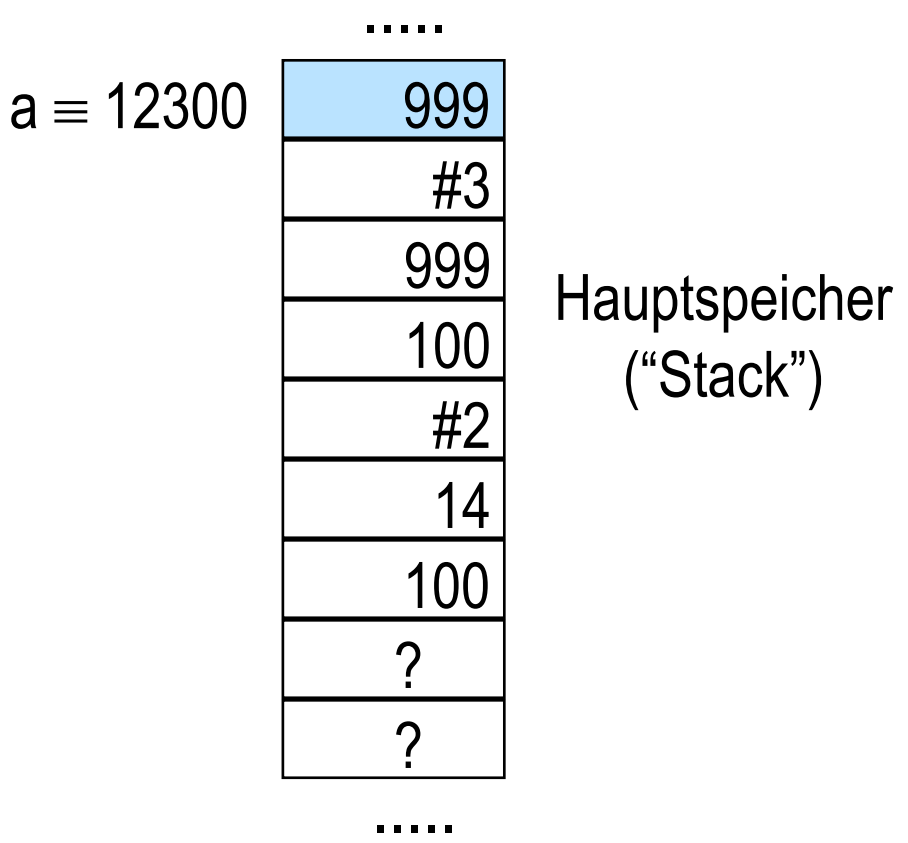
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);

  return 0;
}
```

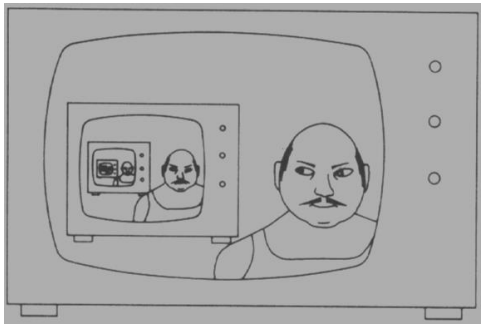


universität wien

Rekursion

• Ein Objekt heißt *rekursiv,* wenn es durch "sich selbst" definiert ist, oder sich selbst (teilweise) enthält.

• Beispiele:
  ◦ Fernseher+Kamera (Droste Effekt)



  ◦ Mathematik (Definition der Fakultät)

$$\forall n \in \mathbb{N}\colon\ n! = \begin{cases} n = 0 \to 1 \\ n > 0 \to n(n-1)! \end{cases}$$

universität wien

Rekursive Programmteile

- Eine Prozedur/Funktion heißt *rekursiv*, wenn sie sich in der Folge ihrer Abarbeitung selbst referenziert.

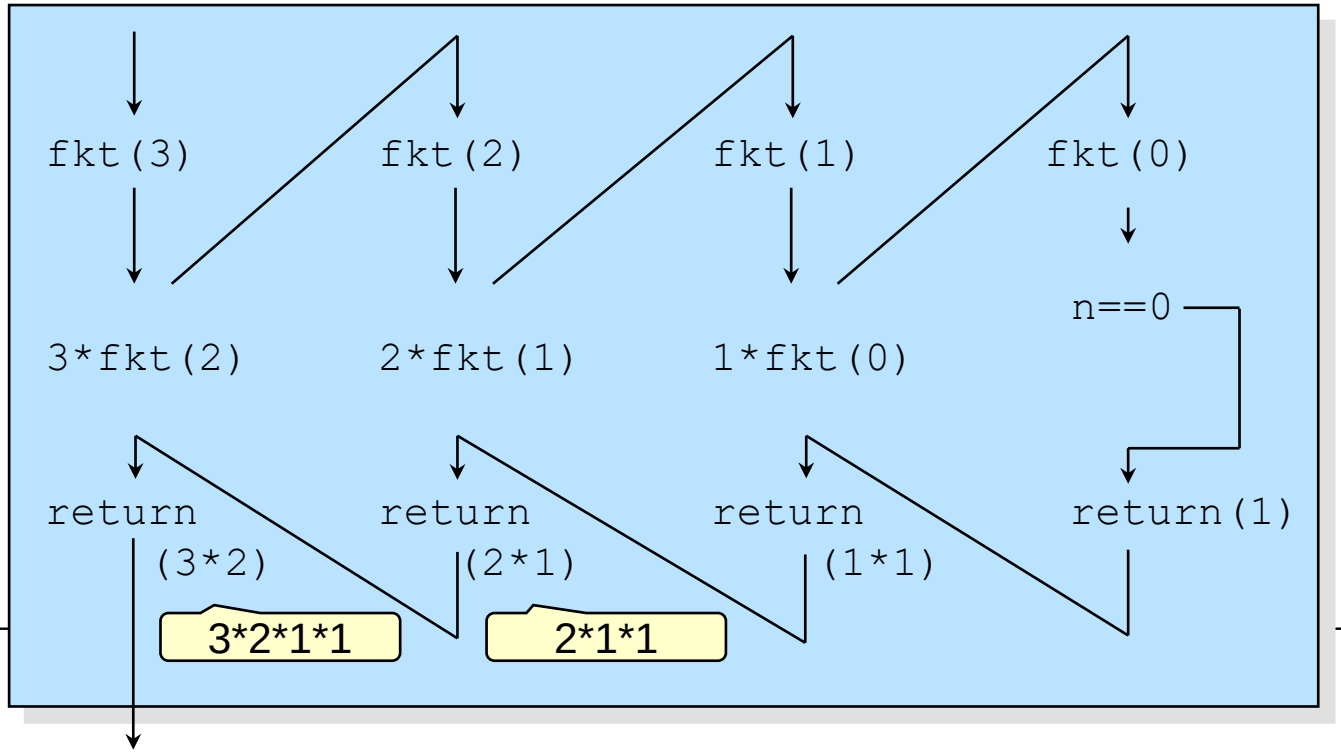
- Eine Prozedur/Funktion P heißt *direkt rekursiv*, wenn sie sich explizit selbst aufruft. Hingegen ist P *indirekt rekursiv*, wenn sie den Aufruf einer anderen Prozedur/Funktion enthält, die ihrerseits wieder P (direkt oder indirekt) aufruft.

Direkte Rekursion: Fakultät

- Berechnung von 3!

  Aufruf: fkt(3) ⇒ 6

```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



universität wien

Direkte Rekursion: Fibonacci Zahlen (1)

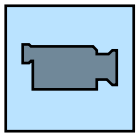
```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

Definition Fibonacci Zahlen:

fibo(1) = 1

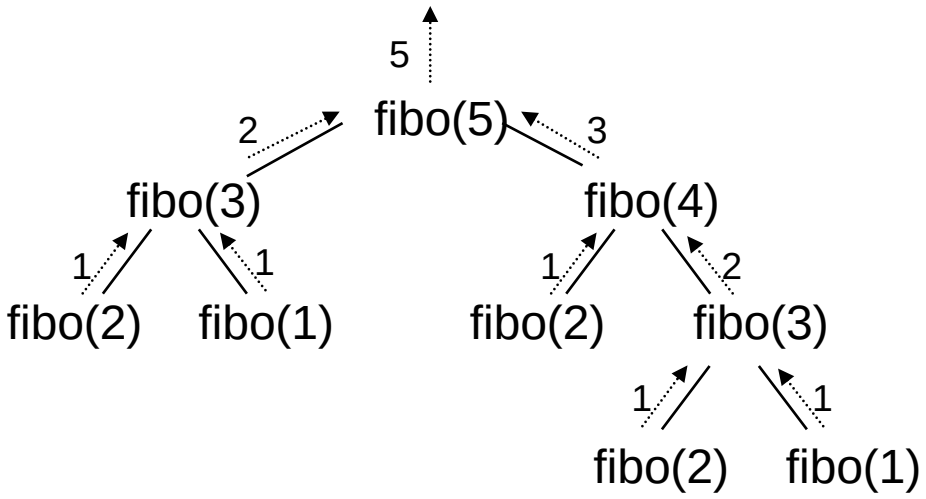
fibo(2) = 1

fibo(n) = fibo(n-2) + fibo(n-1) für n > 2



Direkte Rekursion: Fibonacci Zahlen (2)

- Aufrufgraph




```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return
         fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

universität wien

Wiederholung

• EBNF	`'do'` statement `'while' '(' expression ')'';'`
• Selektion	`if, switch, break`
• Schleife	`while, do while, for, break, continue`
• Logische Ausdrücke	`i=0 || ++i && --i`
• Geltungsbereich	`int i;{int i; {double i; ::i;}}`
• Funktionsdeklaration	`void f(int n);`
• Funktionsdefinition	`void f(int n) {/*...*/}`
• Funktionsaufruf	`f(7)+f(a)*f(a-7);`
• Aufrufmechanismus	Stack, Rücksprungadresse, Parameter, Retourwert
• Rekursion	`void f(int n) {if (n) f(n/2);}`

universität wien

5. Fehlerbehandlung

universität wien

Eine simple Funktion

```
int flaeche (int laenge, int breite){
  return laenge*breite;
}
```

- Offenbar wird die Fläche eines Rechtecks berechnet. Das sollte immer eine positive Größe sein.

- Wir haben aber keine Garantie, dass unsere Funktion nicht mit falschen Parameterwerten aufgerufen wird.

```
flaeche ("anna",3); //wird vom Compiler als ungültig erkannt
flaeche (5.2, 7.9); //OK, Ergebnis "ungenau"
flaeche (7);        //wird vom Compiler als ungültig erkannt
flaeche (-2, 7);    //?
```

universität wien

Wer sollte verantwortlich sein?

- BenutzerIn ist verantwortlich
  - Negative Eingabewerte sind im Handbuch ausdrücklich verboten
    - Wer liest Handbücher?
- Der Aufrufer ist verantwortlich
  - Vor jedem Aufruf müssen die Parameter überprüft werden

```
if (a>=0 && b>=0) flaeche (a, b);
```

    - Mühsam, fehleranfällig, widerspricht der "natürlichen" Faulheit
- Die Funktion ist verantwortlich
  - Prüfung muss nur an einer Stelle im Programm stattfinden
  - Bei ungültigen Parameterwerten muss der Fehler behoben oder ein Fehler signalisiert werden

universität wien

Strategien der Fehlerbehandlung

- Drei prinzipielle Vorgangsweisen:
  - Fehler beheben
    - z.B. nicht erlaubte negative Parameterwerte durch deren Absolutwert ersetzen
  - Fehler ignorieren
    - z.B. Gas geben in einem Auto, das schon mit Höchstgeschwindigkeit fährt
  - Fehler signalisieren
    - Rückmeldung des Fehlers an die aufrufende Funktion

- Die Wahl der Strategie hängt von der genauen Aufgabenstellung ab.
- Im Zweifelsfall Fehler signalisieren.

- Keinesfalls Fehlermeldung ausgeben und das Programm mit wahrscheinlich falschen Ergebnissen weiterarbeiten lassen!

universität wien

Signalisierung von Fehlern

- Retourniere einen speziellen "Fehlerwert"
  - Nicht allgemein verwendbar (es muss zumindest einen Wert geben, der niemals als Ergebnis auftreten kann und daher als "Fehlerwert" nutzbar ist)
  - "Fehlerwert" kann bei späteren Änderungen/Erweiterungen eventuell zu einem legalen Ergebnis werden
- Setze eine (globale) Fehleranzeige

```
int errorno {0};
int flaeche (int laenge, int breite){
  if (laenge<0 || breite<0) errorno = 42;
  return laenge*breite;
}
```

  - Ergebnis (wahrscheinlich falsch) muss trotzdem zurückgeliefert werden

- Beide Varianten verlangen Überprüfung nach jedem Aufruf
  - Hohe Anforderung an die Programmierenden (unrealistisch)
  - Als historische Altlast in einigen Libraries noch immer üblich

universität wien

Exceptions

- Moderne Art, Fehler zu signalisieren
  - Fehlt im Programm der Code, um den Fehler zu behandeln, wird das Programm abgebrochen
  - Fehler können nicht "übersehen" werden!
  - Fehler können an einer zentralen Stelle aufgefangen und behandelt werden
  - Es ist aber immer noch notwendig, genau zu überlegen, wo und wie man Fehler behandelt. (Wirklich simpel ist Fehlerbehandlung nie)

universität wien

Werfen einer Exception (throw)

```
#include<stdexcept> //definiert runtime_error
#include<iostream>
using namespace std;
int flaeche (int laenge, int breite){
  if (laenge<=0 || breite<=0) throw runtime_error("Nur positive Werte erlaubt!");
  return laenge*breite;
}
```

- Es wird ein Objekt vom Typ runtime_error mit einer Fehlermeldung erzeugt und "geworfen"
- Das Objekt wird an die jeweils aufrufenden Funktionen weitergereicht, bis sich eine für diese Art von Fehler (Exception-Typ) für zuständig erklärt und den Fehler behandelt
- Gibt es keinen zuständigen Programmteil (die Exception wandert über main hinaus), wird das Programm abgebrochen

universität wien

Fangen einer Exception (catch)

- Ein Programmteil erklärt sich für bestimmte Arten von Exceptions als zuständig, indem passende Exception-Handler (catch-Statements) angeboten werden

```
try { //falls innerhalb dieses Blocks Exceptions passieren wird ein
      //ein passender catch Block gesucht
  flaeche(7,-3);
}
catch (runtime_error& e) { //Referenz verhindert Kopie – siehe später
  cerr << "Sorry: " << e.what() << '\n';
}
catch (...) { //Alle nicht schon behandelten Exceptions / Reihenfolge wichtig
  cerr << "Ooops, das haette nicht passieren duerfen\n";
}
```

cerr: Vordefinierter Stream zur Ausgabe von Fehlermeldungen. Sehr ähnlich wie cout und gibt ebenfalls standardmäßig auf dem Bildschirm aus.

Anmerkung: Vgl. obige Fehlermeldungen mit "Gib was Gscheites ein, Koffer\n"

universität wien

Traditionell vs. Exception (1)

Traditionell:

```
bool error {false};
…
//Fehler passiert
error = true;
…
if (error) … //Fehlerbehandlung
…
```

Fehleranfällig, da Abfrage nie vergessen werden darf und mühsam in verschachtelten Programmteilen oder bei Funktionsaufrufen

```
bool error {false};
if  …
  if …
     if …
        //Fehler passiert
        error = true;
        …
        if (!error)
          …
     if (!error)
       …
   if (!error)
     …
if (error) … //Fehlerbehandlung
```

Traditionell vs. Exception (2)

Modern mit Exception:

```
try {
  int i {7};
  …
  //Fehler
  throw 3;
  …
  //anderer Fehler
  throw i;
  …
  //noch ein Fehler
  throw 1.2;
}
catch (int& err) {
 …
}
catch (double& d) {
 …
}
catch (...) {
 …
}
```

Programmfluss verzweigt zu einem passenden `catch` Statement am Ende eines (dynamisch) umschließenden `try` Blocks.
Falls kein passendes `catch` Statement vorhanden ist, wird das Programm abgebrochen.

Datentyp muss genau passen. Keine der bisher bekannten Typumwandlungen.

Alle Datentypen (Reihenfolge der catch Statements wichtig).

```
try {
  if  …
    if …
       if …
          //Fehler passiert
          throw 1;
          …
       …
    …
  …
}
catch (int&)… //Fehlerbehandlung
```

Variablenname darf auch weggelassen werden, wenn der Wert zur Fehlerbehandlung nicht benötigt wird.

universität wien

Bis auf Weiteres...

- Einfach alle Exceptions in main abfangen und mit einer anständigen Fehlermeldung Programm beenden

```
#include<stdexcept>

//...

int main (){
  try {
    //...
  }
  catch(...) {
    cerr << "Ooops\n";
  }
  return 0;
}
```

universität wien

preconditions und postconditions

- Preconditions müssen am Anfang einer Funktion erfüllt sein, damit diese ihre Arbeit ordnungsgemäß erfüllen kann (z.B.: Länge und Breite nicht negativ).
  - Kann von ProgrammiererIn der Funktion nicht garantiert werden
  - Möglichst am Anfang der Funktion überprüfen und gegebenenfalls Exception werfen
- Postconditions müssen am Ende der Funktion erfüllt sein. Sie prüfen, ob die Funktion ihre Aufgabe ordnungsgemäß erledigt hat (z.B.: Der berechnete Flächeninhalt darf nicht negativ sein).
  - Nicht erfüllte Postcondition weist auf Programmierfehler hin
  - Exception daher nicht sinnvoll

assert (1)

`assert` hilft, Denkfehler des Programmierers zu finden

```
if (B1)
   if (B2)
      A1;
   else
      // B1 && !B2
      A2;
```

```
#include<cassert>
if (B1)
   if (B2)
      A1;
   else {
      assert(B1 && !B2);
      A2;
   }
```

Korrektheitsbedingung

Bedingung `false`
⇒ Programmabbruch

```
if (B1)
   if (B2)
      A1;
   else ;
else
   // !B1
   A2;
```

```
#include<cassert>
if (B1)
   if (B2)
      A1;
   else ;
else {
   assert(!B1);
   A2;
}
```

```
if (B1) {
   if (B2)
      A1;
} else
   // !B1
   A2;
```

```
#include<cassert>
if (B1) {
   if (B2)
      A1;
} else {
   assert(!B1);
   A2;
}
```

universität wien

assert (2)

```
#include<cassert>
#include<iostream>
using namespace std;
int main() {
   double n;
   cout << "Geben Sie eine Zahl ein: ";
   cin >> n;
   if (n < 0) {
     assert(n < 0);  // Ziemlich überflüssig
     n = -n;
   }
   assert(n > 0);  // Nicht überflüssig!
                   // Deckt einen Denkfehler auf
}
```

Programmaufruf:

```
Geben Sie eine Zahl ein: 0
beispiel.C:12: failed assertion 'n > 0'
Abort
```

universität wien

static_assert

- Seit C++11 ist es möglich, manche Fehler schon zur compile time zu finden

```
'static_assert' '(' bool_constexpr ',' message ')'
'static_assert' '(' bool_constexpr [',' message] ')' //ab C++17
```

- bool_constexpr muss vom Compiler auswertbar sein. Ist der Wert false, wird die Übersetzung abgebrochen und (falls vorhanden) in der Fehlermeldung message angezeigt

universität wien

Postconditions

- Da Postconditions die Programmlogik prüfen, sind assert und static_assert die geeigneten Mittel:

```
#include<stdexcept>
#include<cassert>
#include<iostream>
using namespace std;
int flaeche (int laenge, int breite){  //check precondition
  if (laenge<=0 || breite<=0) throw runtime_error("Nur positive Werte erlaubt!");
  int result {laenge * breite};
  assert(result > 0); //Ist dieses assert sinnlos?
  return result;
}
```

universität wien

Defensive Programmierung

• Bedingungen so weit bzw. eng fassen, dass auch mögliche Sonderfälle korrekt behandelt werden.

```
for (int i {n}; i!=0; --i)
   …
```

„Endlosschleife“ für n<0

```
for (int i {n}; i>0; --i)
   …
```

hält auch für n<0

•Spezialfall reelle Zahlen:

```
for (double x {0}; x!=1; x+=0.1);
//mögliche Endlosschleife

for (double x {0}; !((x>1-eps) && (x<1+eps)); x+=0.1);
//mit einer geeigneten Konstanten eps
```

Nicht
1-eps<x<1+eps

universität wien

Wiederholung

- Fehlerbehandlung `throw, catch`
- Assertions `assert, static_assert`
- Pre- und Post-Conditions
- Defensive Programmierung

universität
wien

6. Die Klassen vector und string

universität wien

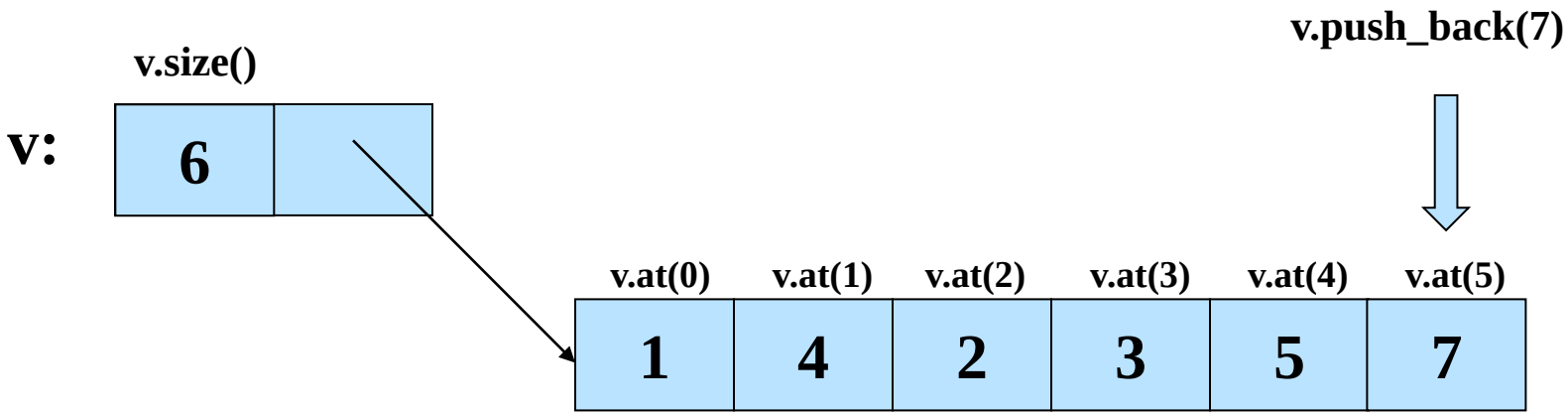
Klasse vector

- Für die meisten Aufgabenstellungen ist es notwendig, eine Menge von Datenwerten zu bearbeiten.
  - Eine Variable für jeden Datenwert zu definieren ist unpraktisch bis unmöglich
  - Die Standardlibrary bietet Datenstrukturen zum Umgang mit Mengen von Werten an
  - Prominenter Vertreter dieser Datenstrukturen ist die Klasse **`vector`**
  - **`vector`** ist eine Templateklasse (man muss also spezifizieren, welche Art von Vektor man haben will, z.B.: **`vector<int>`**)
  - Für Datentypen wie **`vector`** ist die Bezeichnung generische Datentypen (generic datatype) gebräuchlich.

universität wien

Vorstellungsmodell

```
#include<vector>
vector<int> v;
```



universität wien

Methoden

- Wir bezeichnen **`size()`**, **`at()`** und **`push_back()`** als Methoden der Klasse **`vector`**

- Allgemein sind Methoden Funktionen, die eine Klasse zur Verfügung stellt und die für jeweils ein spezielles Objekt der Klasse aufgerufen werden können (`objekt.methode()`)

universität wien

vector und Operator []

- Traditionell wird in C++ mit dem Operator [ ] auf Elemente von Container-Datenstrukturen zugegriffen (vgl. Arrays, siehe später).

- Auch vector bietet [ ] an. Man kann also statt **`v.at(3)`** auch **`v[3]`** schreiben.

- Ein Zugriff auf ein nicht vorhandenes Element (z.B. **`v[-2]`** oder **`v[v.size()]`**) bewirkt aber undefiniertes Verhalten. **`v.at()`** wirft im Fehlerfall eine Exception vom Typ **`std::out_of_range`**.

- Anmerkung: Stroustrup verwendet [ ], aber sichert den Zugriff in seinem Header File entsprechend ab.

universität wien

Traversieren eines Vektors (vector<int>)

```
int sum {0};
for (size_t i {0}; i < v.size(); ++i)
  sum += v.at(i);
cout << sum << '\n';
```

→Anmerkungen

- Eleganter (und oft – besonders für andere Container - auch effizienter) mit range-based for loop (for-each loop)

```
int sum {0};
for (int val: v)
  sum += val;
cout << sum << '\n';
```

- Reihenfolge ist in diesem Fall nicht definiert (es wird in der Regel die, für die jeweilige Datenstruktur effizienteste Reihenfolge verwendet)

universität wien

Traversieren eines Vektors mit unbekanntem Typ

- Eine typunabhängige Formulierung kann sinnvoll sein, wenn derselbe Code mit unterschiedlichen Datentypen arbeiten soll (z.B. bei der Implementierung von generischen Datentypen).

```
for (auto val: v)
  cout << val << ' ';
cout << '\n';
```

- Diese Schleife funktioniert mit Vektoren eines beliebigen Datentyps. Der Datentyp muss nur die Ausgabe mittels Operator << unterstützen.

universität wien

Initialisieren eines Vektors und Zuweisung an Vektor

- Variablen vom Typ **`vector`** können, wie andere Variable auch, initialisiert werden, dabei wird gleich eine ganze Liste von Werten angegeben:

```
vector<int> v {1,7,5,4};
for (int elem : v) cout << elem << " "; //1 7 5 4
```

- Zuweisungen sind ebenfalls auf diese Art möglich.

```
v = {2,8,9,7};
for (int elem : v) cout << elem << " "; //2 8 9 7
```

- Die Listen mit Werten werden als Initialisierungslisten (initializer list) bezeichnet. (Diese können auch beliebig verschachtelt sein, d.h. die Einträge können selbst auch wieder Listen sein usw.)

universität wien

vector: Die wichtigsten Methoden

```
at(pos) //Element an Position pos bzw. Exception
front() //erstes Element bzw. undefiniertes Verhalten
back()  //letztes Element bzw. undefiniertes Verhalten
empty() //true, wenn leer, false sonst

size()  //Anzahl der Elemente

clear() //Vektor leeren

push_back() //Element am Ende anhängen

pop_back() //letztes Element entfernen bzw. undef. Verhalten
```

- Als globale Funktionen (keine Methoden) gibt es außerdem:

```
==, !=, <, >, <=, >= //lexikalische Ordnung
```

- Einfügen/Löschen an beliebiger Position im Vektor ist möglich, dazu werden aber Iteratoren benötigt, die wir erst später behandeln.

universität wien

Sortieren eines Vektors

```
#include <algorithm>
//...



sort(v.begin(),v.end());
```

algorithm bietet viele weitere Funktionalitäten für Container an (in der Regel müssen aber Iteratoren verwendet werden): Mischen, Durchsuchen, Rotieren, Permutieren, Minimum/Maximum Suche ...

universität wien

string

```
#include<string>
std::basic_string<char> //eigentlicher Datentyp
```

- Ähnlich wie Vektor, die Elemente sind aber vom Typ char.
- Zusätzlich zu den Methoden und Funktionen, die auch Vektor anbietet, noch eine Reihe weiterer Möglichkeiten:
- `+, +=    //Konkatenation (+ ist keine Methode)`
- ```
  >>, <<   //Ein-/Ausgabe (keine Methoden)
  length() //Anzahl der Zeichen (Synonym zu size())
  replace(pos,count,str) //Teil von pos bis pos+count-1 durch string str ersetzen
  substr(pos,count)      //Teil des Strings von pos bis pos+count-1
  erase(pos,count)       //Teil des Strings von pos bis pos+count-1 entfernen
  find(str) //Index, an dem str im String auftritt bzw. string::npos
  stoi(), stod(), ..., to_string(val) //Konversionsroutinen (keine Methoden)
  ```
- `...`
  ```

universität wien

string und Zeichenkettenliterale

- **`string`** ist durch implizite Typumwandlungen gut mit Zeichenkettenliteralen verträglich.

- Der Typ eines Zeichenkettenliterals ist aber **`const char[n+1]`**, nicht **`string`**!

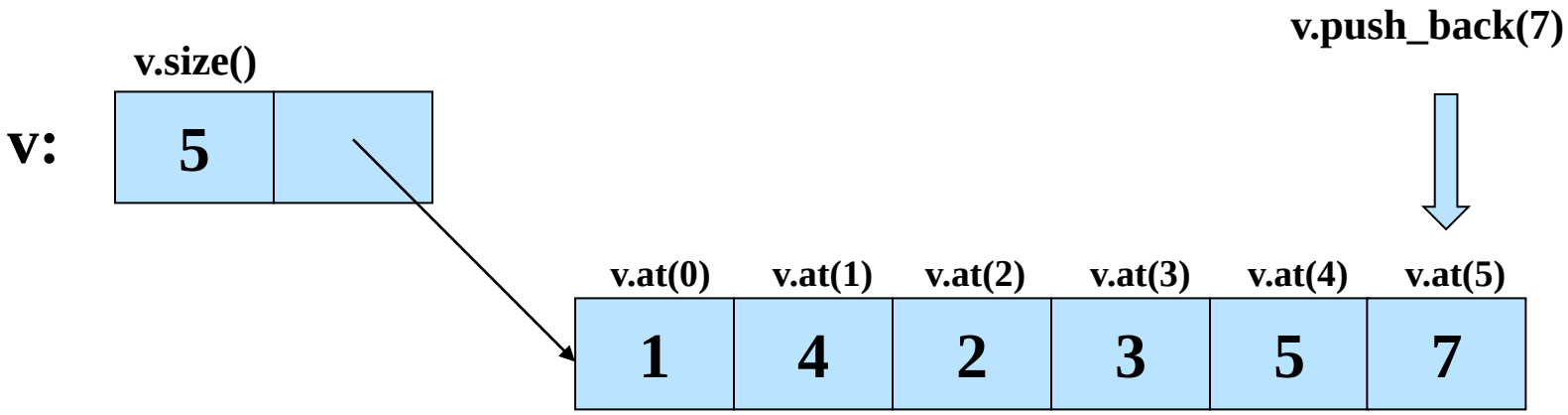
```
string s  {"ab"};    //OK
auto s1 = "cd";       //const char*
s+s1;                 //OK
s+s;                  //OK
//s1+s1; //Compilerfehler, nicht definiert
```

universität wien

Wiederholung

Klasse vector

```
#include<vector>
vector<int> v;
```



Klasse string

Ähnlich wie **`vector<char>`**, allerdings mit vielen zusätzlichen Methoden.
Verträglich mit Zeichenkettenliteralen, aber ein anderer Datentyp!

universität
wien

7. Enumerationen (Aufzähltypen)

universität wien

Aufzähltyp (1)

•Der Aufzähltyp erlaubt es, statt einer *Menge von konstanten Integerwerten* entsprechende Bezeichner zu verwenden:

```
enum Day {sun,mon,tues,wed,thur,fri,sat} aDay;
```

•Die Zuordnung von Werten zu den einzelnen Bezeichnern erfolgt implizit von 0 beginnend der Reihe nach:

```
enum days {sun,mon,tues,wed,thur,fri,sat}aDay;
            0   1   2    3   4    5   6
```

universität wien

Aufzähltyp (2)

- Die Zuordnung kann aber auch explizit erfolgen:

```
enum Karte {zehn=10, bube=2, dame, koenig, as=11};
```

(hier wird keine Variable, sondern nur der Typ **`karten`** und die Konstanten definiert. `dame` erhält den Wert 3 und **`koenig`** den Wert 4 zugewiesen)

- Werte können mehrfach zugeordnet werden, die Konstanten sind dann aber z.B. in einem switch-Statement nicht mehr unterscheidbar

universität wien

Aufzähltyp (3)

- Die Angabe des Namens des Aufzählungstyps kann entfallen:

```
enum {sun,mon,tues,wed,thur,fri,sat} aDay;
```

Es können dann keine weiteren Variablen von diesem Typ vereinbart werden!

- Verwendung: `aDay = mon;`

- Will man *nur* die Konstanten `sun` = 0, `mon` = 1, ... definieren:

```
enum {sun, mon, tues, wed, thur, fri, sat};
```

universität wien

Aufzählтyp implizite Konversionen

- Implizite Konversionen zwischen Aufzählтypen und **`int`**:

```
enum Fuzzy { False, True, Unknown };
Fuzzy b {False};        // O.K.
//b = 0;                // Fehler
int i {False};          // erlaubt
```

- Es gibt keine impliziten Konversionen zwischen unterschiedlichen Aufzählтypen :

```
Fuzzy b;
Karte k {dame};
//b = k;                // Fehler
```

- Explizite Typumwandlungen mit **`static_cast`** sind möglich, führen aber eventuell zu undefiniertem Verhalten.

universität wien

Aufzählтyp Limitierungen

- Die in einer Enumeration definierten Konstanten sind im umgebenden Geltungsbereich gültig. Das kann zu Kollisionen führen:

```
enum Color{red, green, blue};
enum Traffic_light{red, green, yellow}; //Fehler red schon definiert
```

- Der Datentyp, der zur Repräsentation des enum verwendet wird, wird vom Compiler gewählt, das kann in bestimmten Situationen störend sein.

- Beide Limitierungen wurden mit C++11 aufgehoben.

universität wien

Aufzähltyp mit Geltungsbereich (scoped enumeration)

```
enum class Color{red, green, blue};
enum class Traffic_light{red, green, yellow}; //OK
```

- Die Konstanten leben nun in Ihrem eigenen Geltungsbereich.
  Verwendung:

```
Color::red;
Traffic_light::red;
```

- Die Typen sind unterschiedlich und es gibt keine impliziten Typumwandlungen mehr (auch nicht zu int).

```
Color c {green};
Traffic_light t {red};
//c = t;     //Fehler
//int i = t; //Fehler
```

Anmerkung: Statt **`class`** kann auch das Schlüsselwort **`struct`** verwendet werden

universität wien

Aufzählтyp mit Basistyp (type based enum)

```
enum Color : char {red, green, blue};  //unscoped, char based
enum class Weekend : short {sat, sun}; //scoped, short based
```

- Nur für Sonderfälle (z.B.: Kommunikation mit anderen Systemen, besonders effiziente Speicherung) nötig.

- Der vom Compiler gewählte Typ reicht in der Regel aus.

universität wien

Ausgabe von Enum-Werten (1)

- Enum-Werte können nicht direkt ausgegeben werden:

```
cout << Color::red; //Compilerfehler
```

- Man kann die darunterliegende Codierung ausgeben:

```
cout << static_cast<int>Color::red; //gibt den int Wert aus
```

- Oder eine eigene Konvertierungsroutine schreiben:

```
string to_s(Color c) {
  switch (c) {
    case Color::red: return "red";
    //...
  }
}
```

```
int main() {
  cout << to_s(Color::red);
}
```

universität wien

Ausgabe von Enum-Werten (2)

- **`switch`** kann vermieden werden:

```
const vector<string> color_names{"red", "green", "blue"};
cout << color_names.at(static_cast<size_t>(Color::red));
```

- Problematisch, wenn die Codierungen der Enum-Konstanten nicht von 0 beginnend fortlaufend aufsteigend sind
- Die Reihenfolge der Strings muss immer mit der Reihenfolge der Definitionen der Enum-Werte übereinstimmen (dieses Problem lässt sich durch Verwendung von Maps lindern)
- Das kann zu Problemen bei der Erweiterung und Wartung des Programms führen

C++ bietet leider keine überzeugende Möglichkeit für die Ausgabe

---

- Anmerkung: Eingabe von Enum-Werten ist ein analoges Problem

universität wien

Vergleich von Enum-Werten

- Enum-Werte können mit den Vergleichsoperatoren (==, **<**, **>**, etc.) direkt miteinander verglichen werden, wenn sie im selben Geltungsbereich (scope) liegen
- Es werden die darunterliegenden Codewerte verglichen
- Wird die Codierung vom Compiler vorgenommen, ergibt sich somit die Reihenfolge, die durch die Definition der Enum-Konstanten vorgegeben ist

universität wien

Wiederholung

- Enumeration `enum Color {red, green, blue};`
- Werte selbst festlegen `enum Color {red, green=5, blue};`
- Scoped Enumeration `enum class Color {red, green, blue};`
- Type Based Enumeration `enum Color : char {red, green, blue};`
- Implizite Umwandlung zu int nur für unscoped enumerations.

universität wien

8. Referenzen (References)

universität wien

Referenzen

•Referenzen ermöglichen es, einer Variablen einen zusätzlichen Namen zu geben. Eine Referenz wird mittels des Zeichens & definiert und muss initialisiert werden:

```
int i;
int &ir {i};
ir = 7;
cout << i; //Ausgabe: 7
```

•Ermöglicht die Vergabe von Namen für sonst unbenannte Variable

```
vector<int> v {1,3,5};
int &elem {v.at(1)};
cout << elem; //Ausgabe: 3
v.at(1) = 7;
cout << elem; //Ausgabe: 7
```

- Sobald eine Referenz einmal gebunden ist, kann sie nicht mehr verändert werden. Sie verweist immer auf dieselbe Variable.

universität wien

Const Referenzen

- Const Referenzen können zum lesenden Zugriff auf Variable verwendet werden, aber deren Wert nicht verändern.

```
int i {5};
const int &ir {i};
cout << ir; //Ausgabe: 5
//ir = 7;   //nicht erlaubt
```

universität wien

Referenzen als Funktionsparameter

- Bisher haben wir nur Funktionen mit Wertparametern verwendet. Wertparameter werden beim Start der Funktion mit den aktuellen Parameterwerten initialisiert und enthalten dann eine Kopie dieser Werte. Es können keine Seiteneffekte über die Parameter auftreten.

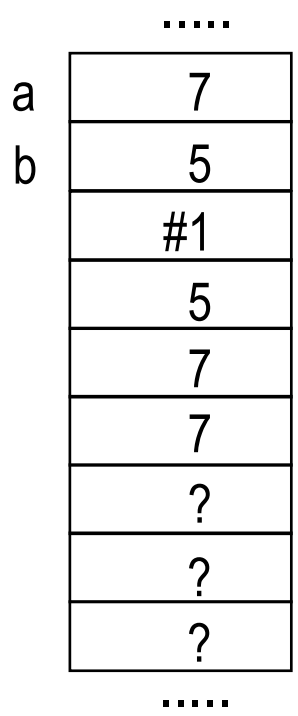
- Manchmal ist es aber wünschenswert, dass eine Funktion die übergebenen Parameterwerte ändern kann:

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
```

Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

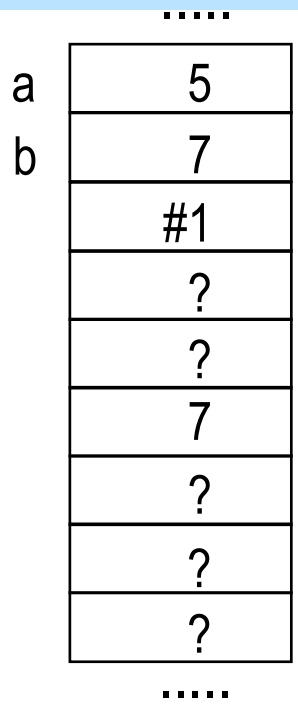
Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



Referenzparameter – call by reference

- Referenzparameter erlauben Seiteneffekte auf die beim Aufruf übergebenen Parameter (diese müssen natürlich Variable sein und nicht Literale)

```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter um Kopien zu vermeiden

- Bei der Übergabe von großen Objekten als Parameter kann die Verwendung von Referenzparametern die Effizienz wesentlich steigern.

```
void f(Film& f, Musik& m);
```

- Um Seiteneffekte zu vermeiden, empfiehlt sich die Verwendung von const-Referenzen.

```
void f(const Film& f, const Musik& m);
```

universität wien

Call by value / reference / const reference

```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr konstant
void g(int a, int& r, const int& cr) { ++a; ++r; int x {cr}; ++x; } // ok

int main()
{
  int x {0};
  int y {0};
  int z {0};
  g(x,y,z);  // x==0; y==1; z==0
  g(1,2,3);  // error: Referenzparameter r braucht Variable zum Referenzieren
  g(1,y,3);  // ok: da cr konstant ist, wird ein "temporäres Objekt" übergeben
}
// const Referenzen für die Parameterübergabe von großen Objekten sehr nützlich
```

Quelle: http://www.stroustrup.com/Programming/lecture-slides.html

universität wien

Referenzen zur Vermeidung von Kopien

- Auch beim Iterieren über die Elemente von Containern und beim Fangen von Exceptions können Kopien eingespart werden:

```
for (const auto& val : v) //const Referenz, wenn Werte nicht
  cout << val << ' ';    //verändert werden sollen
cout << '\n';


try {
//...
}
catch (runtime_error& e) {
//...
}
```

universität wien

Referenz als Returnwert einer Funktion

- Der Returnwert einer Funktion kann ebenfalls eine Referenz sein. Dabei ist aber besondere Vorsicht geboten:

```
int& g() {
  int j;
  return j;  //Referenz wird ungültig (dangling reference)
}            //eventuell Compilerwarnung
int& f(vector<int>& v) {
  if (v.size() < 2) throw runtime_error("Zu kurz");
  return v.at(1); // OK
}
int main() {
  vector<int> v {1,2,3};
  f(v) = 7;
  cout << v.at(1); //7
  int &ref {f(v)};
  //Änderungen im Vektor (z.B. Einfügen/Löschen) können immer noch
  //zur Invalidierung der Referenz führen!
}
```

universität wien

Achtung Falle

- Obwohl das Zeichen & den Datentyp verändert, bindet es syntaktisch mit dem Variablennamen!

```
int i {3};
int &ref1 {i}, ref2 {i};
i = 7;

cout << i << "," << ref1 << "," << ref2 << '\n'; // 7,7,3
```

- In dieser LV verwenden wir als Konvention daher bei der Definition von Variablen die Schreibweise, bei der das &-Zeichen direkt beim Variablennamen steht. An anderen Stellen, wo kein Missverständnis möglich ist, wird das &-Zeichen zum Datentyp geschrieben.

universität wien

Wiederholung

- Referenz
- const Referenz
- Parameter
- Range based for
- catch
- Returnwert
- Syntax

```
int &ref {i};
```

```
const int &ref {i};
```

```
int f(int& p);
```

```
for (const auto& val : container);
```

```
try {} catch (runtime_error& e) {}
```

```
int& f();
```

```
int &ref1{i},ref2{i}; //ref2 keine Referenz!
```

universität
wien

9 Funktionen revisited

universität wien

Funktionen mit Seiteneffekten

- Seiteneffekte in einer Funktion entstehen durch
  - Ein- oder Ausgabe innerhalb der Funktion
  - Zugriff auf globale Variablen
  - Änderung von Referenzparametern innerhalb der Funktion

- Seiteneffekte sind möglichst zu vermeiden, da sie zu allerlei Problemen führen können. Ist ihr Einsatz nötig, so sollte das gut dokumentiert werden.

```
int f(int n) {
  cout << n;
  return n;
}
int g(int& n) {
  return ++n;
}
int main() {
  int n {5};
  cout << f(3) << f(2); //3322 oder 2332?
  cout << g(n) << n; //66 oder 65?
}
```

universität wien

Funktionen ohne Seiteneffekte

- Liefern bei jedem Aufruf mit denselben Parametern immer wieder dasselbe Ergebnis.

- Der Funktionsaufruf kann prinzipiell durch einen beliebigen Ausdruck ersetzt werden, der denselben Wert liefert (Vgl. mathematische Funktionen).

- Diese Eigenschaft wird als *referentielle Transparenz* bezeichnet und ist besonders nützlich bei der mathematischen Modellierung von Programmen (und damit z.B. auch bei den Optimierungen, die der Compiler vornehmen kann).

universität wien

Überladen von Funktionen (function overloading)

- Zwei (oder mehrere) Funktionen dürfen denselben Namen haben, wenn sie sich durch Anzahl oder Datentypen ihrer Parameter (Signatur) voneinander unterscheiden. Der Compiler ordnet einem Aufruf (wenn möglich) die passende Funktion zu. Dies wird auch als (ad hoc) Polymorphismus bezeichnet.

- Beispiel:

```
int max (int a, int b);
double max (double a, double b);
double max (double a, int b);
int max (const int a, const int b);
char max (int i, int j);
max(3,7.5); //ambiguous call
```

`T` und `const T` können beim Aufruf nicht unterschieden werden. `T&` und `const T&` aber schon.

Unterschied nur im Returntyp reicht nicht aus!

universität wien

Defaultargumente -Vorgaben für formale Parameter (1)

• Argumente, die zumeist mit immer dem gleichen Wert belegt werden, können mit Vorgabewerten (Defaults, Standardwerten) belegt werden. Die Vereinbarung von Default-Parametern kann entweder bei der Deklaration oder der Definition erfolgen (ABER nicht gleichzeitig).

• Deklaration:

```
int binomial(int n = 45, int k = 6); //Parameternamen optional
```

• Definition:

```
int binomial(int n = 45, int k = 6) {…}
```

• Aufruf:

```
binomial();              //45 über 6
binomial(10);            //10 über 6
binomial(11,7);  //11 über 7
binomial(,7);            //ist falsch (leider)
```

universität wien

Defaultargumente - Vorgaben für formale Parameter (2)

- Vorsicht: Defaultargumente können nur am Ende der Parameterliste definiert werden!

```
int binomial(int = 45, int);
```

Wird für einen Parameter ein Defaultwert angegeben, so müssen auch alle Parameter rechts davon ebenfalls mit einem Defaultwert versehen werden.

- Da die Reihenfolge, in welcher die Parameter ausgewertet werden, nicht definiert ist, dürfen bei der Initialisierung auch die anderen Parameter nicht verwendet werden!

```
int i;
int f(int i, int j=2*i); //Parameter i zur default Berechnung
```

universität wien

Defaultargumente und Überladen

• Man kann sich die Angabe von Defaultwerten wie die Deklaration/Definition von überladenen Funktionen vorstellen. Technisch wird aber nur eine Funktion deklariert/definiert und fehlende Parameterwerte werden beim Aufruf entsprechend "aufgefüllt".

```
int binomial(int n = 45, int k = 6);
```

entspricht (anschaulich)

```
int binomial();
int binomial(int n);
int binomial(int n, int k);
```

• Weitere Überladungen sind möglich:

```
binomial(double x, double y = 10);
binomial(string cstr);
```

Defaultwerte für `x` bzw. `cstr` hier nicht sinnvoll! Warum?

universität wien

Einschub: weitere Operatoren (1)

- Sequenzoperator (`,`) (sequence operator)
  - Wertet mehrere Ausdrücke in fest definierter Reihenfolge (von links nach rechts) aus. Alle Seiteneffekte des linken Operanden werden abgeschlossen, bevor der rechte Operand ausgewertet wird. Der Wert des gesamten Ausdrucks ist gleich dem zuletzt ausgewerteten Ausdruck.
  - Ist linksassoziativ.
  - Wird meist in `for`-Konstrukten verwendet, wenn zur Initialisierung bzw. Reinitialisierung mehrere Ausdrücke erforderlich sind.
  - Zu unterscheiden von Komma bei Variablendefinition und Parameterliste.

```
int i {0}, j {3};    // Keine Sequenz!
f(1,(j = 2, j), 3); // Drei Parameter (1,2,3)

for (i = j, j = 0; i > 0; i--, j++); //Leeranweisung
cout << i << ',' << j; // Ausgabe: 0,2
```

universität wien

Einschub: weitere Operatoren (2)

- Bedingte Auswertung (`? :`) (conditional expression)
  - Form: Condition `'?'` Ausdruck1 `':'` Ausdruck2
  - Ist rechtsassoziativ, d.h. `x>0 ? 1 : x<0 ? -1 : 0`
    `x>0 ? 1 : (x<0 ? -1 : 0)`
  - Wertet zunächst den booleschen Ausdruck *Condition* aus und schließt dessen Seiteneffekte ab. Falls das Ergebnis true ist, wird *Ausdruck1* ausgewertet, sonst *Ausdruck2*
  - Nur eine Alternative wird ausgewertet
  - Der Wert des gesamten Ausdrucks ist der Wert der ausgewerteten Alternative

```
for (int i {0}; i < 1000; ++i)
  cout << ((i % 10) ? '*' : '\n');
  // 9 Sterne pro Zeile;  Äußere Klammern wichtig
```

universität wien

Einschub: Weitere Operatoren; Seiteneffekte

- Da in C++ die Reihenfolge der Auswertung von Ausdrücken und Funktionsparametern sowie der genaue Zeitpunkt, zu welchem Seiteneffekte wirklich ausgeführt werden, im Allgemeinen nicht fix vorgegeben ist, darf das Programmergebnis nicht von der Reihenfolge der Seiteneffekte abhängen!
- Die neuen Operatoren sind unproblematisch, da Sie eine genaue Reihenfolge der Auswertung der Operanden und der damit verbundenen Seiteneffekte festlegen.

```
x = x++;                    // x wird größer?
x = 0, y = x + ++x;         // y = 1 ??
x = 1, y = (x*=3)+ x;       // y = 4 ??
x = 0, f(x++, x+3);         // f(0,4)??
y = ((x++ * 3)*x++);        // ?? trotz Klammerung
y = x++ && (x % 5);         // OK
y = x++ ? x+=3 : x*2;       // OK
x = 2, x++, x++, y = x;     // OK (y = 4)
```

universität wien

Einschub: weitere Operatoren (3)

- Bitmanipulation:
- Zusätzliche Operatoren für ganze Zahlen, die auf den einzelnen Bits operieren
  - i.a. für systemnahe Programmierung
  - Sind linksassoziativ

- Beispiel:

```
short a {11}; // bei 2 Bytes pro short: 0000000000001011
short b {12}; // bei 2 Bytes pro short: 0000000000001100
cout << (a | b);  //     ODER;  15 =    0000000000001111
cout << (a & b);  //      UND;   8 =    0000000000001000
cout << (a ^ b);  //      XOR;   7 =    0000000000000111
cout << (~a);     // NEGATION; -12 =    1111111111110100
cout << (a << 1); //  L-SHIFT;  22 =    0000000000010110
cout << (a >> 1); //  R-SHIFT;   5 =    0000000000000101
```

auch 1 möglich, Regeln relativ kompliziert

universität wien

Einschub: weitere Operatoren (4)

- Zusammengesetzte Zuweisungsoperatoren (compound assignments)

(**`*=  /=  %=  +=  -=  >>=  <<=  &=  ^=  |=`**)

  - Stellen abgekürzte Schreibweisen für häufig benötigte Zuweisungen dar
  - **`x <op>= y`** ist gleichbedeutend mit **`x = x <op> y`**, außer dass im ersten Fall der Ausdruck **`x`** nur einmal ausgewertet wird
  - Sind rechtsassoziativ wie normale Zuweisung

```
int x {0}, y {1};
y += x += 1;
cout << x << ',' << y; // Ausgabe: 1,2
x *= 4;
cout << x << ',' << y; // Ausgabe: 4,2
(x += y++) += 3;
cout << x << ',' << y; // Ausgabe: 9,3
```

universität wien

inline Funktionen

• Manche Funktionen sind derart simpel, dass der für den Aufruf der Funktion erzeugte Code mehr Laufzeit (und Speicherplatz) kostet als der Code für den Funktionsrumpf:

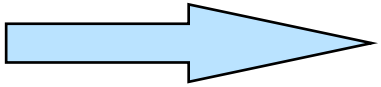
```
double maximum (double a, double b){return a > b ? a : b;}
```

oder

```
double absolut (double x) {return x>=0 ? x : -x;}
```

• Mit dem Schlüsselwort inline wird der Funktionsaufruf, falls möglich, vom Compiler durch die Funktionsdefinition ersetzt:

```
inline double maximum(double a,double b){return a>b ? a:b;}
int main(){
  double x,y,z;
  ...
  z = maximum(x, y);
```



```
x > y ? x : y;
```

• Wird für die Funktion auch eine Deklaration angegeben, dann ist das Schlüsselwort inline sowohl bei der Deklaration, als auch bei der Definition zu verwenden.

universität wien

compile time Funktionen

- Mit dem Schlüsselwort *constexpr* kann man Funktionen definieren, die der Compiler während des Übersetzungsvorgangs bei Bedarf aufrufen kann:

```
constexpr inline double maximum(double a, double b){ //inline optional
  return a>b ? a:b;
}

constexpr double e {2.71};
constexpr double pi {3.14};
constexpr double m {maximum(e,pi)};

int main() {
  int n;
  cin >> n;
  //constexpr double x {maximum(pi, n)};  //n keine constexpr
  double y {maximum(n, e)}; //OK; Aufruf zur Laufzeit
}
```

universität wien

Funktionen und Exceptions (1)

- In Kombination mit Funktionen werden die Stärken des Konzepts der Exceptions offenbar.

- Löst eine Funktion eine Exception aus, so werden alle Funktionsaufrufe beendet, bis ein passendes catch-Statement gefunden (oder das Laufzeitsystem erreicht) wird. Dabei wird der Speicherplatz der lokalen Variablen aller abgebrochenen Funktionen ordnungsgemäß freigegeben (stack unwinding). Dies gilt nicht für dynamisch vom Betriebssystem (mit new bzw. new[ ] – siehe später) angeforderten Speicher.

- Als Bild für die Funktionsweise von Exceptions kann man sich eine Befehlskette vorstellen (Funktion ≡ Person; Funktionsaufruf ≡ Befehl; Exception ≡ Rückmeldung, falls Befehl nicht ausgeführt werden kann).

universität wien

Funktionen und Exceptions (2)

```
void g() {
  int i {3};
  cout << "g1 ";
  throw i;
  cout << "g2 ";
}

void f() {
  int i;
  cout << "f1 ";
  g();
  cout << "f2 ";
}

int main() {
  int i;
  cout << "main1 ";
  try {
    f();
    cout << "f done ";
  }
  catch(int& err) {}
  cout << "main2 ";
}
```

Es wird eine Kopie von i angelegt und als Exception gesendet. Warum?

```
main1 f1 g1 main2
```

Fangen der Exception mittels Referenz, vermeidet neuerliche Kopie und ermöglicht Änderungen an der Exception (z.B. um sie dann weiterzureichen)

universität wien

Rethrow

```
void f() {
  int i {3};
  try {
    throw i;
  }
  catch(int& err) {
    err = err + 1;
    throw;
  }
}
int main() {
  int i;
  try {
    f();
  }
  catch(int& err) {
    cout << err;
  }
}
```

```
4
```

Rethrow: Die ursprünglich empfangene Exception wird neuerlich geworfen

universität wien

noexcept - Spezifikation

• Eine Exception – Spezifikation legt fest, ob eine Funktion eine Exception werfen darf

```
void f() noexcept;
void g() noexcept(condition);
```

• Ergibt die condition true oder fehlt sie, so darf die Funktion keine Exception nach außen dringen lassen. Andernfalls wird das Programm beendet.

Die zuvor eingeführten Exception-Spezifikationen (throw(...) Klauseln) sind seit C++11 deprecated und wurden mit C++17 aus dem Standard entfernt. throw() als ‚Alternative zu noexcept wurde mit C++20 aus dem Standard entfernt.

universität wien

Wiederholung

- Funktion mit Seiteneffekten
- Überladen
- Defaultparameter
- Sequenzoperater
- Bedingte Auswertung
- Bitmanipulation
- Inline Funktion
- Compile time Funktion
- noexcept Spezifikation

```
cout << f(x) << x; //eventuell undef.
void f(int); int f(double); void f();
int f(int = 2, double = 1.2);
i=2, ++i, i*=j
return a<b ? a : b;
~ << >> & ^ | //absteigende Priorität
inline void f(int);
constexpr int f(double);
void f(int) noexcept;
```

universität wien

10 Klassen

universität wien

Motivation

- Immer wieder treffen wir auf Werte, die logisch zusammen gehören, z.B.: x- und y-Koordinaten von Punkten im $\mathbb{R}^2$ oder Vor-, Zuname und Geburtsdatum einer Person.

- Solange wir nur mit einzelnen Punkten oder Personen arbeiten, ist das nicht weiter problematisch. Wenn wir aber viele derartige Dinge verarbeiten wollen, wird das schnell unübersichtlich. z.B.:

```
vector<double> points_x;
vector<double> points_y;
```

Keine Garantie, dass die Vektoren gleich lang sind.
Beim Einfügen und Löschen darf man keinen der Vektoren vergessen.

universität wien

Strukturen

- Ermöglichen es, Datenelemente zu einem neuen Datentyp zusammenzufassen:

```
struct Point {
  double x, y;
};

vector<Point> points;
```

Eine Stelle, wo in C++ ein ; nach einer } notwendig ist.

- Eine Struktur stellt einen benutzerdefinierten Datentyp dar. Konventionsgemäß beginnen die Namen von benutzerdefinierten Datentypen mit einem Großbuchstaben.

- Ausnahme: Die Namen der Datentypen in der Standard Library (vector, string, etc.)

universität wien

Verwendung

```
Point p {4, 1}; //komponentenweise Initialisierung
Point q;
q = p; //komponentenweise Zuweisung
q.x *= 2; //Zugriff auf Komponente mit . Operator
```

- Benutzerdefinierte Datentypen sind den eingebauten Datentypen sehr ähnlich. Insbesondere können sie auch wieder in anderen benutzerdefinierten Datentypen als Typen von Komponenten auftreten:

```
struct Rectangle {
  Point lower_left, upper_right;
  Color color; //Color anderswo definiert
  int border_width;
};
```

universität wien

Objekte

- In der objektorientierten Programmierung werden reale Gegebenheiten (Objekte oder Konzepte) durch Objekte im Programm repräsentiert.

- Objekte haben einen Zustand und ein Verhalten.

- Der Zustand eines Objekts kann in den Variablen eines benutzerdefinierten Typs repräsentiert werden. Für das Verhalten können diese Typen um Funktionen erweitert werden, z.B.:

```
struct Circle {
  Point center;
  double radius;

  void scale(double factor) {radius *= factor;}
};
```

universität wien

Verhalten von Objekten

```
Circle c {{0,1}, 2};
c.scale(2); //hat jetzt Radius 4
c.radius = -1; //?
```

- Sehr oft sind die möglichen (gültigen) Zustände von Objekten eingeschränkt.

- Objekte müssen Integritätsbedingungen (integrity constraints) erfüllen. Diese werden oft auch als Invarianten (invariant) bezeichnet.

- Um garantieren zu können, dass die Integritätsbedingungen immer erfüllt sind, muss der Zugriff auf die Instanzvariablen von außen eingeschränkt werden. Das ist das Prinzip der Datenkapselung (data hiding).

universität wien

Klassen

```
class Circle {
  Point center;
  double radius;

public:
  void scale(double factor) {radius *= factor;}
};
```

privat, von außen nicht zugreifbar

ab hier von außen zugreifbar

- Mit den Schlüsselworten private, public und protected kann die Sichtbarkeit von Klassenmitgliedern (member) festgelegt werden:
  - private: Nur innerhalb der Klasse sichtbar
  - public: Überall sichtbar
  - protected: Innerhalb der Klasse und in von ihr erbenden Klassen (siehe später) sichtbar
- Die Schlüsselworte private, public und protected können beliebig oft und in beliebiger Reihenfolge verwendet werden.

universität wien

class vs. struct

- Einziger Unterschied:
  - struct startet implizit mit public:
  - class startet implizit mit private:

- Faustregel: struct nur verwenden, wenn es keine Einschränkungen für die Gültigkeit der Objekte gibt. Sonst ist class zu bevorzugen.

universität wien

Terminologie

- Die Klasse beschreibt, wie die Zustände der Objekte repräsentiert werden (Instanzvariablen – instance variables, Eigenschaften – properties, Attribute – attributes) und wie das Verhalten der Objekte gesteuert werden kann (Methoden – methods, Operationen – operations).

- Objekte sind Ausprägungen (Instanzen – instances) einer Klasse.Sie besitzen eine Identität und einen aktuellen Zustand (Werte der Instanzvariablen).

- Instanzvariablen sind Variablen, die in jedem Objekt (in jeder Instanz) vorhanden sind. Sie werden in der Klasse deklariert und beim Erzeugen eines neuen Objekts angelegt. Zugriff mittels objekt.variablenname (soweit die Instanzvariable sichtbar ist).

- Methoden sind Funktionen, die für ein Objekt aufgerufen werden mittels objekt.methodenname(parameterliste) (soweit die Methode sichtbar ist).

universität wien

Trennung von Interface und Implementierung

- Das Interface enthält alle notwendigen Informationen zur Verwendung der Klasse.
- Die Implementierung enthält den eigentlichen Code.
- Trennung erlaubt, die Implementierung zu ändern / korrigieren / optimieren, ohne die Clientprogramme (jene, die die Klasse verwenden) zu beeinflussen (das gilt natürlich nur so lange, wie keine Änderungen am Interface durchgeführt werden müssen).

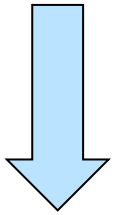
universität wien

Aufteilung in unterschiedliche Dateien

```
#include "point.h"
class Circle {
  Point center;
  double radius;

public:
  void scale(double);
};
```

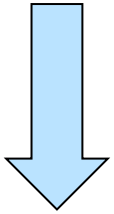
Deklaration typischerweise
in einer Header Datei (h, hpp)



circle.h

```
#include "circle.h"
void Circle::scale(double factor) {
  radius *= factor;
}
```

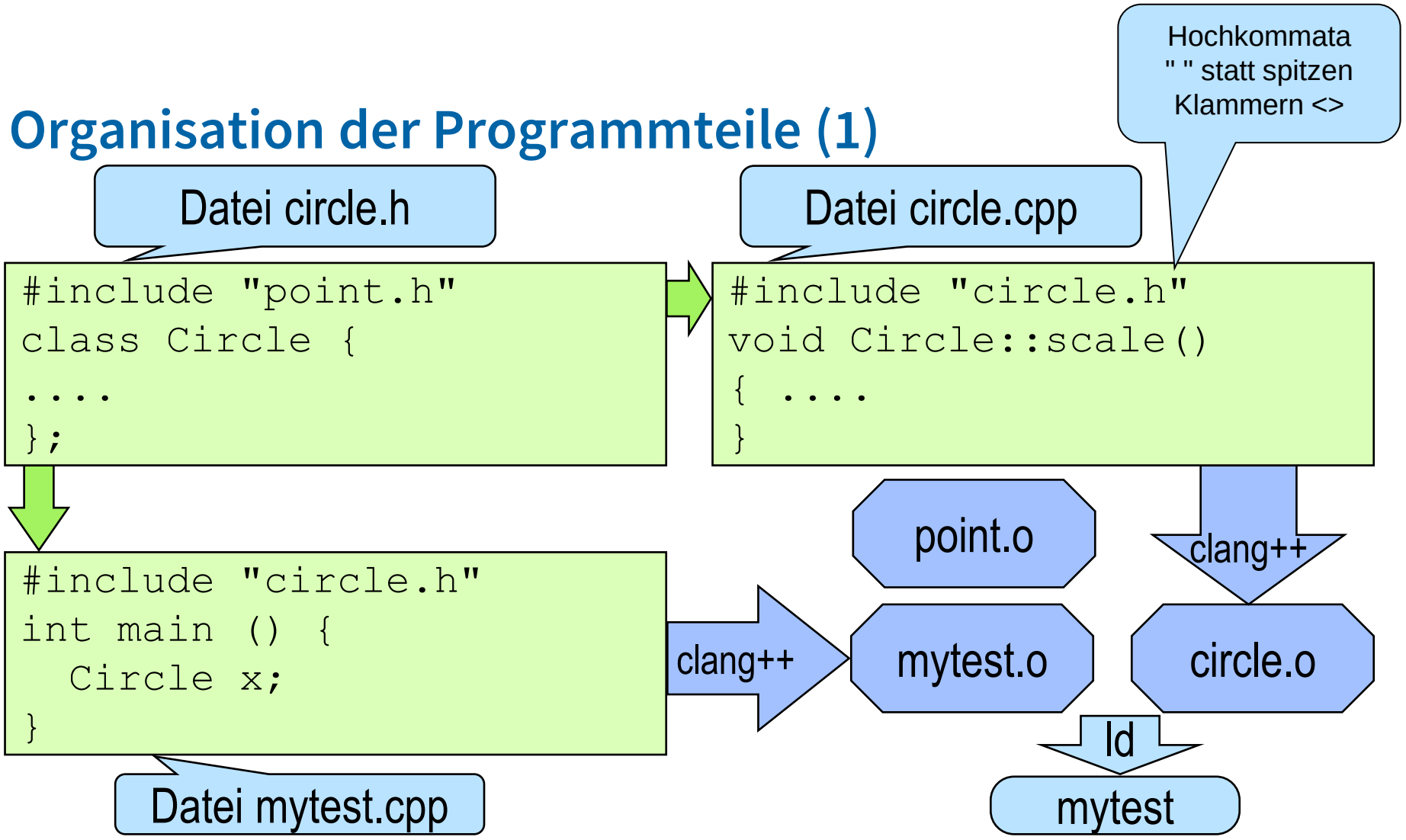
Definition (Implementierung) typischerweise in einer cpp (cc, C) Datei



circle.cpp

universität wien

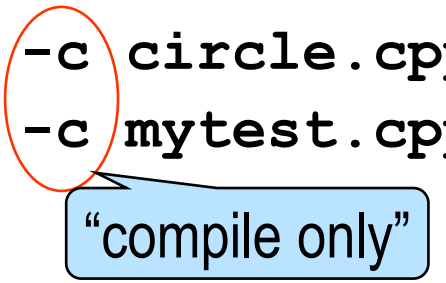
Organisation der Programmteile (1)



universität wien

Organisation der Programmteile (2)

- Erzeugung des exekutierbaren Programms (test):
  - **`clang++ -c circle.cpp`** (übersetzen, liefert circle.o)
  - **`clang++ -c mytest.cpp`** (übersetzen, liefert mytest.o)



  - **`clang++ mytest.o circle.o point.o -o mytest`** (binden)

- Alternative:
  - **`clang++ mytest.cpp circle.cpp point.o -o mytest`** (übersetzen & binden)

universität wien

Verhindern, dass eine Datei mehrmals inkludiert wird

```
#pragma once

#include "point.h"
class Circle {
  Point center;
  double radius;

public:
  void scale(double);
};
```

Nicht im Standard, aber von vielen Compilern unterstützt

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "point.h"
class Circle {
  Point center;
  double radius;

public:
  void scale(double);
};

#endif
```

Die klassische Methode

universität wien

Namespaces

- Um Namenskollisionen zu vermeiden, können Namensräume verwendet werden:

```
#pragma once
#include "point.h"

namespace Geometric_figures {
class Circle {
  Point center;
  double radius;

public:
  void scale(double);
};

class Triangle; //forward Deklaration wird später definiert
}               //als class Geometric_figures::Triangle
```

universität wien

Verschachtelung von Namespaces und Hinzufügen von weiteren Namen

```
#pragma once
#include "point.h"

namespace Geometric_figures {
  namespace Plane_figures {
    //...
  }
  namespace 3D_figures {
    //...
  }
}

namespace Geometric_figures { //in derselben oder einer
       //...                  //anderen Datei
}
```

universität wien

Using Declarations und Using Directives

- Das Schreiben eines voll qualifizierten Namens (fully qualified name) kann auf die Dauer mühsam sein:

```
std::cout << "Hello world\n";
```

- Die using directive erleichtert das erheblich, allerdings werden unter Umständen unerwünschte Namen mit importiert (z.B. min/max oft störend):

```
using namespace std;
cout << "Hello world\n";
cin >> x;
```

- Mit einer using declaration lässt sich ein einzelner Name importieren:

```
using std::cout;
cout << "Hello world\n";
std::cin >> x;
```

universität wien

Zugriff auf Objekte in Methoden (1)

- Eine Methode darf die private members aller Objekte ihrer Klasse, sowie die protected members aller Objekte der Basisklasse und die public members aller Objekte aller Klassen verwenden.
- Da eine Methode immer mit einem Objekt aufgerufen wird, hat sie dieses Objekt (das quasi als eine Art zusätzlicher impliziter Parameter an die Methode übergeben wird) automatisch im Zugriff. Der Objektname darf beim Zugriff auf members des aktuellen Objekts entfallen, so weit der Zugriff eindeutig ist:

```
void Circle::scale(double factor) {
  radius *= factor;  //radius des aktuellen Objekts
  translate(); //Methodenaufruf für das aktuelle Objekt
}
```

universität wien

Zugriff auf Objekte in Methoden (2)

- Wird ein anderes als das aktuelle Objekt benötigt, muss wie sonst auch der Zugriff mittels **`objektname.membername`** erfolgen

```
//Schnitt zweier Kreise
Vector<Point> Circle::cut(const Circle& c) {
  if (center != c.center) { //nicht konzentrisch
    //... extrem viel komplizierter Mathe-Krams
  }
  else {
    if (radius == c.radius)
      throw runtime_error("Unendlich viele Schnittpunkte");
    else
      throw runtime_error("Keine Schnittpunkte");
  }
}
```

universität wien

Zugriff auf Objekte in Methoden (3)

- Manchmal ist es aber notwendig, das aktuelle Objekt mit einem Namen anzusprechen. Dafür gibt es das Schlüsselwort `this`:

```
Circle Circle::test(double radius) {
  if (this->radius > radius) //Zugriff auf member mit ->
    this->translate();  // this eigentlich nicht nötig
  return *this; //Zugriff auf Objekt selbst mit *
}
```

universität wien

Zugriff auf Objekte von außen

- Mittels getter- und setter-Methoden (auch Akzessoren – accessors und Mutatoren – mutators) kann auch von außen auf die privaten Instanzvariablen zugegriffen werden:

```
double Circle::get_radius() {return radius;}

void Circle::set_radius(double radius) {
  if (radius <= 0) throw runtime_error("negativer Radius");
  this->radius = radius;
}
```

- Die Kontrolle bleibt immer innerhalb der Klasse und diese kann somit die Integritätsbedingungen garantieren.

universität wien

Initialisierung von Objekten

- Wie für alle anderen Variablen auch, sind die Werte von Instanzvariablen zunächst einmal undefiniert. Das ist bei Objekten besonders störend, da dadurch vielleicht bereits die Integritätsbedingungen nicht erfüllt sind. Das Objekt startet schon "kaputt" ins Leben.

```
Circle c; //eventuell hat c negativen Radius
```

- Es gibt daher spezielle Methoden, die immer bei der Erzeugung eines neuen Objekts aufgerufen werden, die sogenannten Konstruktoren.

universität wien

Konstruktoren (1)

• Bei der Erzeugung von Instanzen (Objekten) einer Klasse wird eine spezielle Methode (“Konstruktor”) aufgerufen, die die neue Instanz initialisieren (d.h., ihre Instanzvariablen mit sinnvollen Werten belegen) soll.
Diese Methode heißt genauso wie die Klasse, hat keinen Rückgabewert (auch nicht void) und beliebig viele Parameter, mit denen die Initialisierungswerte festgelegt werden können.

• Deklaration:

```
Circle(const Point& = Point{0,0}, double = 1);
Circle(double);
```

Implizite Deklaration von `Circle()` dem **Defaultkonstruktor** zur Initialisierung von Default-(Standard-) Objekten. Wird bei Bedarf (und wenn gar kein Konstruktor in der Klasse definiert ist) vom Compiler automatisch erzeugt.
(Dieser automatisch erzeugte Defaultkonstruktor führt keine speziellen Initialisierungen von Instanzvariablen durch.)

universität wien

Konstruktoren (2)

- Implementierung

```
Circle(const Point& center, double radius): center{center}, radius{radius} {
  if (radius<=0) throw runtime_error("Not a circle");
}

Circle(double r) : Circle{Point{0,0},r} {}
```

member initializer list: außerhalb der Klammer steht eine Instanzvariable, innerhalb der Klammer gilt scope des Funktionsblocks (hier: Parameter)

Delegating constructor / Constructor chaining

Hier wird eventuell eine Exception ausgelöst. Wird ein Konstruktor via Exception verlassen, dann gilt das Objekt als nicht existent. (Das Objekt „lebt“ vom Ende des Konstruktors bis zu seiner Vernichtung, wenn die zugehörige Variable "out of scope" geht)

universität wien

Konstruktoren (4)

- Verwendung

```
int main() {
  Circle c1;                  //Defaultkonstruktor (Einheitskreis)
  Circle c2 {100};            //Mittelpunkt (0,0), Radius 100
  Circle c3 {Point{10,15}}; //Mittelpunkt (10,15), Radius 1
  Circle c4 {Point{-1,3},7};//Mittelpunkt (-1,3) Radius 7
}
```

universität wien

Konstruktoren und Typumwandlung

• Konstruktoren mit genau einem Parameter können auch als Typumwandlung angesehen werden und vom Compiler bei Bedarf zur impliziten Typumwandlung verwendet werden.

```
void f(Circle);
f(Point{0,0});
```

- Wenn es keine Funktion **f** mit einem Parameter vom Typ **Point** gibt, wird implizit von **Point** auf **Circle** umgewandelt.
- Damit das nicht unübersichtlich wird, wendet der Compiler pro Parameter maximal eine implizite Typumwandlung mittels Konstruktor an.

• Die implizite Anwendung des Konstruktors kann durch Verwenden des Schlüsselworts **explicit** verhindert werden. Danach kann der Konstruktor nur mehr direkt oder durch explizite Typumwandlung aufgerufen werden.

```
//explicit nur in der Deklaration
explicit Circle(Point = Point{0,0}, double = 1);
Circle c{Point{0,0}}; //OK
c =static_cast<Circle>(Point{0,0}); //cast - OK
f(Point{0,0});    //keine implizite Umwandlung - nicht erlaubt
```

universität wien

Konversionsoperatoren

• Ein Konstruktor mit einem Parameter kann zur Konversion eines beliebigen Datentyps in ein Objekt der Klasse verwendet werden. Will man eine Konversion in der anderen Richtung definieren, kann man Konversions-Operatoren definieren (explicit ist auch hier möglich):

```
class Circle{
   …
   operator int();
   operator AndereKlasse();
   …
}
```

Einzige Möglichkeit der Konversion in einen Standarddatentyp

Konversionsoperator hat keinen Returntyp (auch nicht void) und keine Parameter. Warum?

Wird verwendet, wenn kein entsprechender Konstruktor in der anderen Klasse erstellt werden kann

universität wien

Überladen von Operatoren (1)

- Fast alle C++ Operatoren können überladen werden. Dabei bleiben Arität, Bindungsstärke und Assoziativität unverändert, aber die Semantik der Operation kann neu definiert werden. Nur :: ?: . .* sizeof typeid alignof noexcept können nicht überladen werden.
- Missbrauch vermeiden!

- Zum Überladen eines Operators wird eine globale Funktion oder Methode erstellt, deren Namen sich aus dem Wort operator und dem jeweiligen Operatorsymbol zusammensetzt (z.B.: operator+)

- Der erste (linke und für unäre Operatoren einzige) Operand wird im Falle einer Methode immer als this-Objekt übergeben. Eine Operatormethode hat somit immer einen Parameter weniger als der Operator Operanden hat. Eine globale Operatorfunktion hat immer gleich viele Parameter wie der Operator Operanden hat.

- Einige Operatoren (=, (), [], ->) können nur mit Methoden überladen werden. Bei Postfix-Inkrement und -Dekrement wird zur Unterscheidung von den analogen Prefix-Operatoren ein zusätzlicher (ansonst redundanter) Parameter vom Typ int verwendet.

universität wien

Überladen von Operatoren (2)

```
class Circle {
  …
  bool operator==(const Circle& c) const {//Methode für binären Operator ==
    return (center == c.center && radius == c.radius);
  }
  …
};
bool operator!=(const Circle& c1, const Circle& c2) {
//globale Funktion für binäres !=
  return !(c1==c2);
}
```

- Aufruf:

Wegen impliziter Typumwandlung möglich. (Solange Konstruktor nicht explicit ist.)

```
main() {
  Circle c1, c2;
  Point{0,0} != c1;
  c1 == Point{0,0};        //OK
  //Point{0,0} == c1;      //Error
  operator!= (c1,c2);      //möglich, aber eher ungewöhnlich
  c1.operator== (c2);      //möglich, aber eher ungewöhnlich
}
```

universität wien

Überladen von Operatoren: Methode oder globale Funktion

- Üblicherweise werden Methoden bevorzugt, weil man innerhalb der Methode direkten Zugriff auf die Instanzvariablen hat.

- Bei Operatoren ergibt sich aber, wie auf der vorigen Folie gezeigt, eine Asymmetrie zwischen this-Objekt und Parameter. Um das zu vermeiden, werden oft globale Funktionen für das Überladen von Operatoren bevorzugt. Wird für diese Operatorfunktionen direkter Zugriff auf Instanzvariable benötigt, kann das Schlüsselwort **`friend`** (siehe später) verwendet werden.

universität wien

Überladen von Operatoren: Spezielles

• Der Returntyp und die genauen Typen der Parameter (Wert, Referenz, konstante Referenz) sind prinzipiell frei wählbar. Ausnahme ist operator->, das einen Pointer oder ein Objekt für das wiederum operator-> definiert sein muss, retournieren muss. Für operator= gelten ebenfalls spezielle Regeln (siehe später: Kopierzuweisungsoperator, copy assignment).

• Für &&, || und , gehen beim Überladen die Regeln der Ausarbeitungsreihenfolge verloren (bis C++17). Für überladene Operatoren && und || werden immer beide Operanden in nicht spezifizierter Reihenfolge ausgewertet. Ab C++17 werden die Operanden der überladenen Operatoren &&, || und , von links nach rechts ausgewertet (allerdings nur, wenn der Operator verwendet wird und nicht die Funktion - z. B. operator&&(a,b) - explizit aufgerufen wird).

friend-Funktionen / -Methoden / -Klassen

- Mittels des Schlüsselworts `friend` kann eine Klasse globalen Funktionen oder Methoden anderer Klassen den Zugriff auf private Instanzvariable erlauben.

```
class Circle{
  …
  friend bool operator== (const Circle& lop, const Circle& rop);
  friend void Class::method();
  friend class Other;
  …
};
```

Deklaration der **weiterhin globalen Funktion** `operator=` als friend der Klasse Circle

Nur die Methode method der Klasse Class hat Zugriff

Alle Methoden der Klasse Other haben Zugriff

universität wien

const Methoden

• Durch Anfügen des Schlüsselworts const an den Funktionskopf (sowohl bei der Deklaration, als auch bei der Definition) wird das aktuelle Objekt (this) als konstant definiert.

```
class Circle{
  …
  bool foo (int) const;
  …
};
bool Circle::foo (int index) const {…}
```

• Innerhalb der Methode **`foo`** darf das this-Objekt nun nicht mehr verändert werden. Die Methode darf aber dafür auch für konstante Objekte aufgerufen werden.

• const ist „ansteckend“. Der Aufwand zahlt sich aber im Hinblick auf korrekte und leichter wartbare Programme im Allgemeinen aus.

universität wien

Funktionen vs. Methoden

(„globale“) Funktion

Methode

Deklaration (Prototyp)

```
C add (C op1, C op2);
```

```
class C { /* ... */
        C add (C op2);
};
```

„gehört“ zu jedem C-Objekt

Definition (Implementation)

```
C add (C op1, C op2) {
/* ... op1.real + op2.real ... */
}
```

```
C C::add (C op2) {
/* ... real + op2.real ... */
}
```

Aufruf

```
z = add(x, y);
```

```
z = x.add(y);
```

Operatorfunktion/-methode

```
C operator+ (C op1, C op2) {...}
```

```
C C::operator+ (C op2) {...}
```

Aufruf

```
z = x + y;
z = operator+(x, y);
```

```
z = x + y;
z = x.operator+(y);
```

universität wien

Spezialfall: Ausgabeoperator (stream insertion)

- Mit **`cout << ...`** können unterschiedliche Objekte ausgegeben werden, weil der Operator **`<<`** (mehrfach) überladen ist

```
class ostream … { … public: …
    ostream& operator<<(int n);
    ostream& operator<<(long n);
    ostream& operator<<(double n);
    ostream& operator<<(char c);
    ostream& operator<<(const char *s);
};
```

Auszug aus der Datei iostream

„Durchschleifen“ des „Orts“ der Ausgabe (z.B. **`cout`**, **`cerr`**), daher Folgendes möglich:

```
int n; char c;
cout << n << c << '\n';
```

Eigentlich: **`(((cout << n) << c) << '\n');`**

universität wien

Ausgabe von Klassenobjekten

- Zwei prinzipielle Möglichkeiten, um Ausgabe mit **<<** zu erreichen:

- Typkonversion: Erstellen einer Methode zur Konversion in einen Datentyp, der schon ausgegeben werden kann, etwa

```
Circle::operator string() {…}
```

- Weiteres Überladen des Operators **<<**: Die überladene Methode **`operator<<`** müsste zur Klasse **`ostream`** hinzugefügt werden (da der linke Operand immer als this-Objekt übergeben wird und somit die Klasse festlegt, in der die Methode verwirklicht werden muss). Die Veränderung der vorgegebenen Bibliotheken ist aber (wenn überhaupt möglich) keine gute Idee!

universität wien

operator<< (1)

- Operator **<<** wird daher als globale Funktion überladen

```
ostream& operator<< (ostream& o, const Circle& c);

ostream& operator<< (ostream& o, const Circle& c) {
    …    // c ausgeben
    return o;
}
```

Ausgabe nicht so einfach, weil Zugriff auf die Instanzvariablen für die globale Funktion verboten ist. Man könnte die globale Funktion operator<< als friend deklarieren

universität wien

operator<< (2)

- Als Alternative zur Definition von operator<< als `friend` bietet sich die Verwendung von Accessoren oder für den jeweiligen Zweck speziell zu implementierenden Methoden an.
- Ausgabe ist sowieso langsam, daher ist die Effizienz nicht relevant. Es kann eine (noch zu erstellende) `print`-Methode verwendet werden.

```
ostream& operator<< (ostream& o, const Circle& s) { //kein friend
  return s.print(o);
}
```

Da s eine Referenz auf einen konstanten Wert ist, darf print sein this-Objekt nicht verändern. print muss daher eine const-Methode sein. print erhält den Stream, auf den ausgegeben werden soll, als Parameter und liefert konventionsgemäß diesen Stream auch als Returnwert.

universität wien

operator << (3)

```
ostream& Circle::print(ostream& o) const {
  return o << "[(" << center.x << ", " << center.y << "); " << radius << ']';
}
```

Alle Ausgaben an das gewünschte Ziel, nicht an `cout`

universität wien

inline Methoden

• Einfache Methoden (Komponentenfunktionen) können auch als inline-Funktionen formuliert werden:

Explizit durch Schüsselwort **`inline`** (sowohl bei Deklaration als auch bei Definition)

```
class Circle{
   …
   inline Circle(double r);
   inline ostream& print(ostream &);
   …
};

inline Circle::Circle(double r):
                Circle{Point{0,0},r} {}

inline ostream& Circle::print(ostream&) {
 …
}
```

Implizit (ohne Angabe des Schlüsselwortes **`inline`**) durch Definition innerhalb der Klassendefinition

```
class Circle {
   …
   Circle(double r):Circle{Point{0,0},r} {}
   …
   ostream& print(ostream&) {
      …
   }
   …
};
```

universität wien

Klassenvariablen

- Instanzvariablen existieren für jede Instanz einmal. Gibt es keine Instanzen einer Klasse, so gibt es auch keine Instanzvariablen.
- Klassenvariablen existieren für jede Klasse genau einmal. Sie beschreiben Eigenschaften der Klasse (aller Objekte der Klasse).

```
class KDTest { static unsigned no; … };    // Deklaration
unsigned KDTest::no{0};      // Definition außerhalb, ohne static!
```

- Seit C++03 können ganzzahlige Klassenkonstanten bei der Deklaration initialisiert werden. Sie können dann in konstanten Ausdrücken (z.B. für die Definition der Größe eines Arrays) verwendet werden. Für eine andere Verwendung ist wieder eine Definition (dann aber ohne Initialisierung) notwendig.

```
class KDTest { static constexpr size_t size{80}; … };
```

- Ab C++17 können ganzzahlige Klassenkonstanten beliebigen Typs mithilfe des Schlüsselworts inline definiert und initialisiert werden.

```
class KDTest {inline static const vector<string> v{"a", "b", "c"}; … };
```

universität wien

Klassenmethoden

• Analog zu Klassenvariablen gehören Klassenmethoden zur Klasse (sie haben kein this-Objekt und können deshalb ohne ein Objekt aufgerufen werden; Auf Instanzvariable kann daher nur unter Spezifizierung eines Objekts zugegriffen werden.):

```
class KDTest {
   static unsigned no;
public:
   static unsigned readCounter() { return no; };
};
```

Verwendung: `KDTest::readCounter();`

oder auch: `KDTest k; k.readCounter();`

Objekt `k` dient nur zur Festlegung der Klasse. Eventuell wird eine Compilerwarnung generiert.

universität wien

Wiederholung

- Klassen
- Instanzvariablen
- Klassenvariable (Deklaration)
- Zugriffserlaubnis
- Methoden
- Konstruktoren
- Typumwandlung
- Operator überladen
- const-Methoden
- this
- inline-Methoden
- Klassenmethode
- Klassenvariable (Definition)

```
class Circle {
   Point center; double radius;
    static String desc;
public:
   ostream& print();
   Circle(const Point& = Point{0,0}, double = 1);
   explicit Circle(string);
   operator int ();
   Circle& operator[](int index) const {
         …
         return *this;
  }
  static int get_desc();
};
String C::desc="Description";
```

universität
wien

11 Iteratoren

universität wien

Iterieren über die Elemente eines Vektors

```
vector<int> v {1,2,3,4};
int sum {0};
```

Die zwei uns bereits bekannten Varianten:

- 1) Mit Hilfe der Methoden size() und at() der Klasse vector:

```
for (size_t i {0}; i < v.size(); ++i) sum += v.at(i);
```

- 2) Eleganter mit range-based for loop (for-each loop):

```
for (int elem: v) sum += elem;
```

- Die Variante 2 ist nur möglich, da vector Iteratoren unterstützt!

universität wien

Iterieren über beliebige Container

- Viele Container bieten keinen indizierten Zugriff und/oder speichern die Daten nicht zusammenhängend im Memory.

- Iteratoren sind ein Zugriffskonzept. Ein Iterator referenziert dabei einen Wert in einem Container. Ein Iterator kann weitergeschaltet werden um auf den nächsten Wert zu referenzieren. Dafür werden zumindest die beiden Operatoren * (Zugriff, „Dereferenzieren“) und ++ (Prefix Inkrement; Weiterschalten zum nächsten Wert) angeboten. Weitere Operationen sind optional. Im Kontext mit range-based-for-loops ist != (Vergleich zweier Iteratoren) notwendig (meist wird dann auch gleich == angeboten).

- Der C++ -Standard definiert einige Iterator-Typen, die unterschiedliche Operationen anbieten. Es ist genau festgelegt, welche Operationen ein bestimmter Typ anbieten muss. Die Details sind komplex.

universität wien

Methoden begin() und end()

•Ein Container, der mit Iteratoren durchlaufen werden kann, bietet in der Regel die Methoden begin() und end() an, die die beiden benötigten Iteratoren retournieren. (Manchmal werden auch globale Funktionen verwendet.)

- Traversieren direkt mit Iteratoren:

```
for (vector<int>::iterator it {v.begin()}; it != v.end(); ++it) sum += *it;
```

- Da der Datentyp der Iteratoren je nach Container unterschiedlich ist, verwenden wir **`auto`**:

```
for (auto it{v.begin()}; it != v.end(); ++it) sum += *it; //v wird durchlaufen
```

Ab C++17 OK. Davor (wegen **`auto) it = v.begin()`** verwenden!

universität wien

Iterator-Typen (1)

- In der Praxis werden meist mindestens zwei unterschiedliche Iterator-Typen benötigt. Einer zum Zugriff auf nicht konstante Objekte (lesend und schreibend) und einer zum Zugriff auf konstante Objekte (nur lesend).

- Für Klassen gibt es const- und nicht-const-Varianten von begin() und end() mit passenden Returntypen.

- Je nach Container können außer den Basisfunktionen auch noch weitere Möglichkeiten angeboten werden. Etwa operator-- um sich in der anderen Richtung bewegen zu können (bidirectional iterator) oder wahlfreier Zugriff mit operator[]. Manche Container können nur einmal durchlaufen werden (z.B. Streams) andere beliebig oft. Manche Container können nur lesend oder nur schreibend zugegriffen werden.

universität wien

Iterator-Typen (2)

<table>
<tr><td colspan="5">Iterator category</td><td>Defined operations</td></tr>
<tr><td rowspan="5">ContiguousIterator</td><td rowspan="4">RandomAccessIterator</td><td rowspan="3">BidirectionalIterator</td><td rowspan="2">ForwardIterator</td><td>InputIterator</td><td>•read
•increment (without multiple passes)</td></tr>
<tr><td></td><td>•increment (with multiple passes)</td></tr>
<tr><td colspan="2"></td><td>•decrement</td></tr>
<tr><td colspan="3"></td><td>•random access</td></tr>
<tr><td colspan="4"></td><td>•contiguous storage</td></tr>
<tr><td colspan="6">Iterators that fall into one of the above categories and also meet the requirements of OutputIterator are called mutable iterators.</td></tr>
<tr><td>OutputIterator</td><td colspan="4"></td><td>•write
•increment (without multiple passes)</td></tr>
</table>

Quelle: http://en.cppreference.com/w/cpp/iterator

universität wien

Iterator Zustände

- Obwohl die Operationen Dereferenzieren und Inkrement immer angeboten werden müssen, kann deren Anwendung auf einen Iterator in bestimmten Zuständen illegal sein. Wir unterscheiden daher folgende Zustände:

- Dereferenzierbar (dereferencable): Der Iterator kann dereferenziert werden und liefert einen Wert. Der von end() gelieferte Iterator ist nicht dereferenzierbar, ebenso wie eventuell ein Input- oder Output-Iterator, der bereits dereferenziert wurde (single-pass).

- Inkrementierbar (incrementable): Der Iterator kann zum nächsten Wert weitergeschaltet werden. Der von end() gelieferte Iterator ist nicht inkrementierbar. Ein Output-Iterator könnte beispielsweise immer abwechselnd dereferenzierbar und inkrementierbar sein.

- Anwenden von dem Zustand nicht entsprechenden Operationen führt zu undefiniertem Verhalten!

universität wien

Invalidierung (Invalidation) von Iteratoren

- Durch Operationen am Container wie Löschen oder Einfügen können Iteratoren ungültig (invalidiert) werden. Invalidierte Iteratoren sind weder dereferenzierbar, noch inkrementierbar.

- Welche Operationen bei welchen Containern zur Invalidierung von (bestimmten) Iteratoren führen, ist in der Dokumentation der jeweiligen Operationen zu finden.

- Im Anhang findet sich eine kurze Übersicht.

universität wien

Bereich (Range)

- Durch Angabe zweier Iteratoren kann ein Bereich festgelegt werden. Ein Bereich ist nur gültig, wenn beide Iteratoren valid sind und durch fortgesetztes Inkrementieren des ersten Iterators irgendwann einmal der zweite Iterator erreicht wird.
- Ungültige Bereiche führen bei Verwendung zu undefiniertem Verhalten!
- Die Methoden begin() und end() liefern bei STL-Containern einen Bereich, der alle Elemente des Containers umfasst.
- begin() liefert einen Iterator, der das erste Element referenziert, end() einen Iterator, der das – virtuelle – Element nach dem letzten Element referenziert.
- Damit ist es möglich auch leere Bereiche darzustellen, begin() == end() und beide referenzieren auf das virtuelle Endelement. Daher darf end() nicht dereferenziert werden!

universität wien

Range-Based-For-Loop

- Bietet ein Container Iteratoren mit zumindest den Operationen ++ (Prefix), * und != an, sowie die Methoden begin() und end() an, dann kann dieser Container mittels einer range-based-for-loop iteriert werden. Da der Datentyp der Iteratoren je nach Container unterschiedlich ist, verwenden wir auto.

```
for (auto it{v.begin()}; it != v.end(); ++it) ...
```



```
for (const auto& elem : container) ...
```

universität wien

Erase und Insert von vector (1)

- Die Methoden erase und insert erhalten einen Iterator (seit C++11 einen const_iterator), der die Position angibt. Dabei löscht erase genau den Wert, auf den der Iterator verweist und insert fügt vor dem referenzierten Wert ein.

- Beide Methoden invalidieren alle Iteratoren (sowie Pointer und Referenzen), die auf Werte nach der Einfüge- bzw. Löschposition verweisen. Beim Einfügen können sogar alle Iteratoren (sowie Pointer und Referenzen) invalidiert werden, wenn es zu einer Allokation eines größeren Speicherbereichs kommt.

- Beide Methoden retournieren einen Iterator, der auf das eingefügte Element oder auf das Element, das auf das gelöschte Element folgt, verweist. Wurde das letzte Element gelöscht, so wird end() retourniert.

universität wien

Erase und Insert von vector (2)

- Das Nichtbeachten der Invalidierung von Iteratoren durch erase() bzw. insert() kann zu schweren Fehlern führen!

```
for (auto& elem : v) {
  ...
  erase bzw. insert Aufruf für v
  ...
}
```

```
for (auto it{v.begin()}; it!=v.end(); ++it) {
  if (*it==0) v.erase(it);
}
```

```
for (auto it{v.begin()}; it!=v.end();) {
  if (*it==0) it = v.erase(it);
  else ++it;
}
```

universität wien

Erase und Insert von vector (3)

```
vector<int> v;
v = {1, 4, 9};
```

- Element 4 soll gelöscht werden. erase() von vector liefert einen Iterator, der auf das darauf folgende Element referenziert oder end(), zurück.

```
for(auto it{v.begin()}; it!=v.end();) {
  if(*it==4) it = v.erase(it);
  else ++it;
}
```

- Konkret wird also die Referenz auf 9 (3tes Element vor dem Löschen) von erase zurückgeliefert. Daher dürfen wir nur dann den Iterator weiterschalten wenn nicht gelöscht wurde. Der Schleifendurchlauf für 9 würde sonst entfallen. Wäre 4 das letzte Element würde sogar eine Endlosschleife entstehen da end() übersprungen werden würde.

universität wien

Erase und Insert von vector (4)

```
vector<int> v;
v = {1, 4, 9};
```

- Element 6 soll vor Element 4 eingefügt werden. insert von vector liefert hier jedoch einen Iterator, der auf das eingefügte Element selbst referenziert, zurück.

```
for(auto it{v.begin()}; it!=v.end();) {
  if(*it==4) {
    it = v.insert(it, 6);
    ++it;
  }
  ++it;
}
```

- Konkret wird also die Referenz auf 6, das eingefügte Element (2tes Element nach dem Einfügen, vor 4) zurückgeliefert. Zusätzlich weiterschalten ist notwendig (auf 9 bzw. end() falls 4 bereits das letzte Element war) da sonst das Element 4 erneut gefunden und endlos eingefügt werden würde.

universität wien

Algorithmen (1)

- Die C++ Standard Library bietet Funktionen die vordefinierte Algorithmen implementieren die häufig benötigte Funktionalität für Container zur Verfügung stellen. Sie haben in der Regel einen Bereich, oder Iteratoren als Parameter.

- Durch die Verwendung von Iteratoren sind diese Algorithmen unabhängig von den darunterliegenden Containertypen.

- Suchen, Sortieren, Mischen, Minimum, Maximum, ...

- Manche Algorithmen verlangen spezielle Typen von Iteratoren (z.B. bidirektional oder multi-pass).

universität wien

Algorithmen (2)

- Ein Beispiel ist das Finden des Minimums in einem Bereich.

```
auto it{min_element(v.begin(),v.end())}; //Minimum in v suchen
```

- Der Returnwert ist ein Iterator, der das gefundene Minimum referenziert, er ist gleich v.end(), wenn es kein Minimum gibt. Da der genaue Typ des Iterators vom zugrundeliegenden Container abhängt, wurde hier wieder auto (und keine initializer list) verwendet.

universität wien

Funktionen als Parameter

• Bei der Formulierung von Algorithmen ist es durchaus nicht ungewöhnlich, dass Funktionen als Parameter verwendet werden.

```
auto i{find_if(v.begin(),v.end(),predicate)};
```

- Soll einen Iterator liefern, der das erste Element in einem Bereich referenziert, für das der Aufruf der Funktion predicate true liefert.

- Es gibt mehrere Methoden, um Funktionen als Parameter zu definieren:
  - 1) Funktionsobjekte/Funktoren (Function objects/Functors)
  - 2) Lambdaausdrücke (Lambda expressions)
  - 3) Funktionszeiger (Function pointer)

- Diese Konzepte werden später noch vorgestellt.

universität
wien

12 Die Klasse unordered_map

universität wien

Klasse unordered_map

- Eine Unordered Map kann, wie auch ein Vektor, eine Menge von Datenwerten verwalten. Eine Unordered Map bietet aber die Möglichkeit über einen Schlüssel einen Datenwert zu finden.

- **`unordered_map`** ist eine Templateklasse (man muss also spezifizieren, welche Datentypen der Schlüssel und der Wert haben,
  z. B.: **`unordered_map<int, string>`**)

- Durch die Verwendung von Schlüsseln sind die Suche, das Einfügen und das Löschen sehr effizient verglichen zu der Klasse vector (abhängig von der Datenmenge).

- Die Klasse unordered_map bietet, wie auch die Klasse vector, die Methoden **`size()`**, **`at()`** und **`operator[]`** an.

universität wien

Methode insert

- Die Methode insert() ermöglicht das Einfügen eines Datenwerts. Dabei müssen jedoch Schlüssel und Datenwert als Paar übergeben werden.

```
#include<unordered_map>
unordered_map<int, string> fruechte;
fruechte.insert(pair<int, string>{1,"Apfel"});
fruechte.insert(pair<int, string>{2,"Birne"});
fruechte.insert(pair<int, string>{3,"Ban"});
//Spezifikation des Typs pair<int, string> ist seit C++11 optional
```

- Die Datentypen (Templateparameter) des struct pair müssen jenen der Unordered Map entsprechen (bzw. Implizit in diese umgewandelt werden können).

- insert() erlaubt auch andere Parameter, mit denen z.B. ganze Bereiche über Iteratoren eingefügt werden können.

universität wien

Methoden at und operator[]

- **`operator[]`** wie auch **`at()`** mit dem Schlüssel als Parameter liefert eine Referenz auf den Datenwert, dieser kann daher verändert werden. Die Datentypen sind dabei jeweils die Typparameter der Deklaration der Unordered Map (hier ist der Schlüssel vom Typ **`int`**, der Datenwert vom Typ **`string`**).

```
string& apfel{fruechte[1]};
apfel = "Granny Smith";
string& birne{fruechte[2]};
birne = "Williamsbirne";
```

- Existiert der Schlüssel nicht, so wird ein Wert mit diesem Schlüssel eingefügt und eine Referenz auf den eingefügten (leeren) Datenwert zurückgeliefert (**`operator[]`**) bzw. eine Exception vom Typ **`std::out_of_range`** geworfen (**`at`**-Methode).

universität wien

Methode find

- **`find()`** liefert einen Iterator (Datentyp **`unordered_map<int, string>::iterator`**) zur Vereinfachung kann auto verwendet werden.

```
auto banane = fruechte.find(3);
```

- Der Zugriff über den Iterator auf Schlüssel und Datenwert erfolgt mittels **`operator*`** bzw. **`operator->`**. Dabei referenziert der Iterator jedoch ein Datenpaar von Schlüssel und Datenwert. Die Instanzvariablen first und second erlauben dann den Zugriff auf Schlüssel und Datenwert.

```
(*banane).second += "an"; //Klammern unbedingt notwendig
banane->second += "e";    //Alternative Syntax ohne Klammern
```

- Datentyp von (**`*banane`**) ist hier **`pair<int, string>`**. Die Datentypen entsprechen wieder den beiden Templateparametern der Deklaration der Unordered Map.

universität wien

Methode erase

- **`erase()`** benötigt einen Schlüssel als Parameter und liefert als Ergebnis die Anzahl der gelöschten Werte.

```
fruechte.erase(2);
```

- Alternativ kann auch ein Iterator als Position übergeben werden.

```
auto frucht = fruechte.find(3);
fruechte.erase(frucht);
```

- Ohne auto wäre hier die genaue Typangabe erforderlich:

```
unordered_map<int, string>::iterator f{fruechte.find(3)};
fruechte.erase(f); //undefiniertes Verhalten!
```

- Eine weitere Überladung erlaubt durch Übergabe eines Anfangs- und Enditerators das Löschen eines ganzen Bereichs.

universität wien

Traversieren einer Map

- Das Traversieren einer Unordered Map ist analog zum Vektor. Der Iterator referenziert wie auch beim find() auf Datenpaare über deren Instanzvariablen first und second auf die Schlüssel und Datenwerte zugegriffen werden kann.

```
for(const auto& frucht : fruechte)
  cout << "Schlüssel: " << frucht.first <<
          ", Wert: " << frucht.second << '\n';
```

13. Felder, Zeiger u. dynamischer Speicher

universität wien

Felder (1)

- In einem Feld (Array) werden mehrere Objekte gleichen Typs zusammengefasst. Die einzelnen Feldelemente werden über ihre Positionsnummer (Index) angesprochen.
- Der Index ist eine ganze Zahl zwischen 0 und (Elementanzahl-1).
- Definition:

```
Datentyp Name '[' Anzahl ']' {'[' Anzahl ']' } ';'
```

```
int vektor[3];
double matrix[3][3];
int kubus[10][10][10];
```

- Felder werden von C++ als Datentyp zur Verfügung gestellt. Im Unterschied von vector oder anderen Container-Klassen, die in Libraries ausprogrammiert sind.

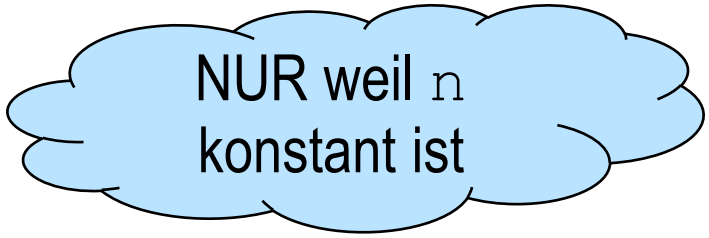
universität wien

Felder (2)

- Zugriff:

  Name '[' Index ']' {'[' Index ']' }

```
constexpr int n {1};
double wert[ 2*n ];
wert[0] = 1.0;
wert[n] = 2.0;
```



- Initialisierung von Feldern durch Angabe einer Werteliste in geschwungenen Klammern

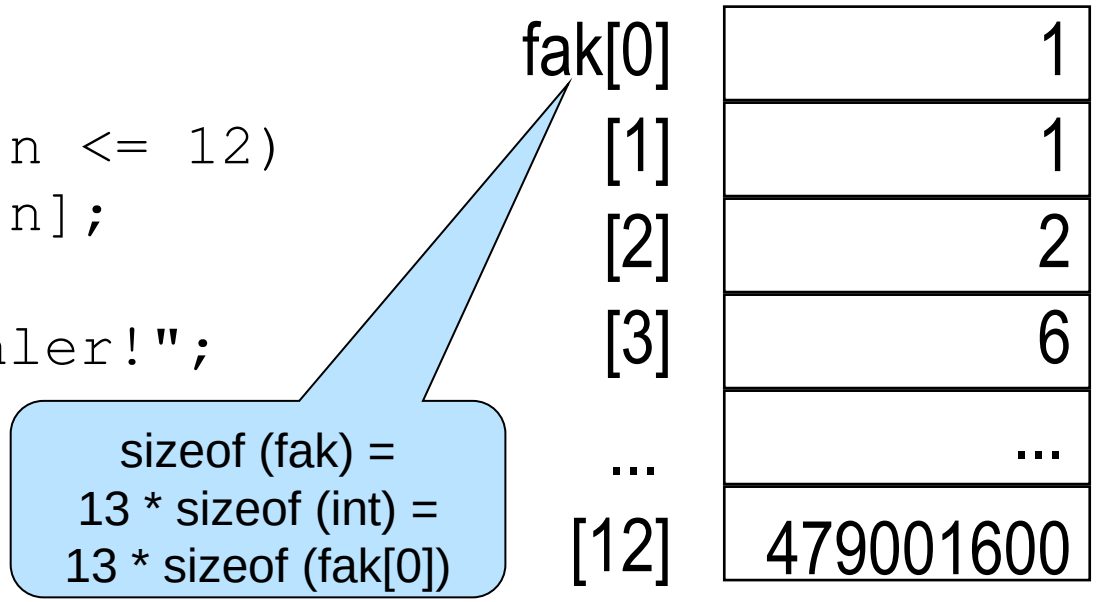
```
double ex[3] { 1, 0, 0 };
double ey[3] { 0, 1 };    // drittes Element implizit 0
double ez[]  { 0, 0, 1 }; // Länge des Feldes wird aus der
                          // Initialisierungsliste ermittelt
```

universität wien

Felder (3)

```
int main () {
  int fak[13] {1,1,2,6,24,120,720,5040,40320,
               362880,3628800,39916800,479001600};

  int n;
  cin >> n;
  if (n >= 0 && n <= 12)
    cout << fak[n];
  else
    cout << "Fehler!";
}
```



universität wien

Felder (4) – Zeichenketten

- Zeichenketten (Strings) sind spezielle Felder: **"C++"** ist ein **char**-Feld der Länge 4.
- Zeichenketten werden durch ein Nullbyte (**'\0'**) terminiert, d.h. die physische Ausdehnung ist um eins größer als die logische
- Initialisierung von Zeichenketten durch Zeichenkettenliterale möglich

```
char s[] {"C++"};
```

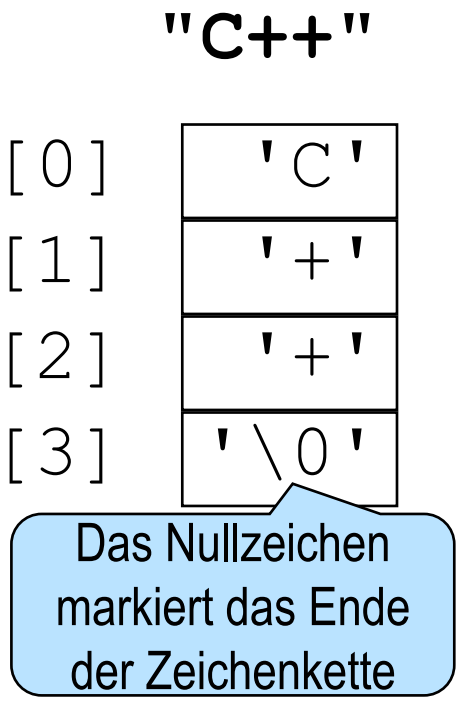
- folgende Anweisung ermittelt die (logische) Länge einer Zeichenkette s:

```
for (i=0;s[i]!='\0';i=i+1);
```

Leeranweisung als Schleifenrumpf

- Direkte Ein- und Ausgabe wird für Zeichenketten (nicht für andere Felder) unterstützt:

```
cout << s << "abc";
cin >> s;
```



universität wien

Felder (5) – Zeichenketten

- Textzeile einlesen und verkehrt wieder ausgeben:

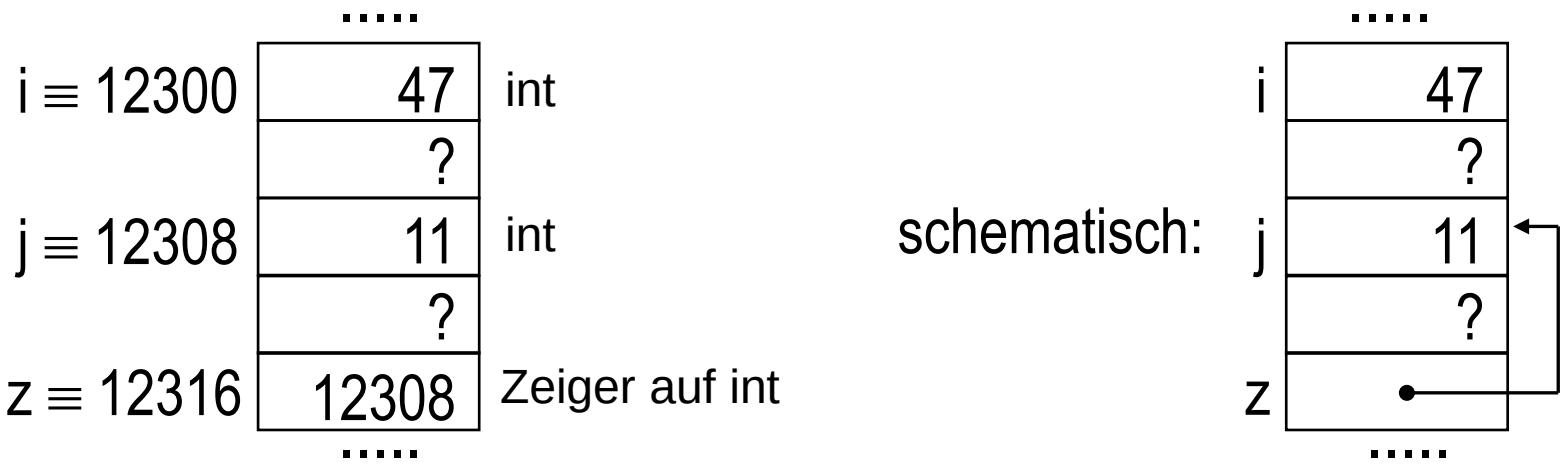
```
#include<iostream>
using namespace std;
int main() {
  char zeile[80];
  int i, n;
  do {
    cin >> zeile;
    for (n = 0; zeile[n]!='\0'; ++n);
    for (i = n-1; i>=0; --i) {
      cout << zeile[i];
    }
    cout << '\n';
  } while (1);
}
```

Achtung: Endlosschleife,
Abbruch durch ^C (Ctrl-C)

universität wien

Zeiger (1)

- Zeiger (Pointer) werden für den indirekten Zugriff auf Daten verwendet.
- Ein Zeiger enthält die Adresse des Speicherplatzes, an dem das Datenelement zu finden ist.

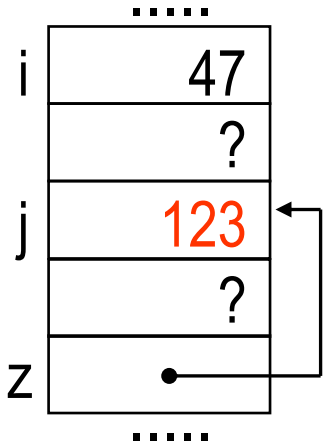


- Lesen/Zuweisen eines Wertes mittels eines Zeigers ergibt/verändert den Wert der Variable, auf die gezeigt wird.

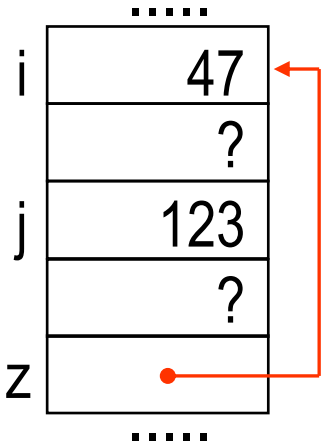
universität wien

Zeiger (2)

• Bei Zeigern gibt es (im Unterschied zu normalen Variablen) zwei verschiedene Manipulationsmöglichkeiten (Zuweisungen):



Verändern/Lesen des Werts, auf den gezeigt wird.



Verändern(“Verbiegen”)/Lesen des Zeigers (der Adresse) selbst.

universität wien

Zeiger (3)

Falle: * gehört eigentlich zum Datentyp, bindet aber syntaktisch mit dem Namen

- Zeigervariable werden vereinbart, indem in einer "normalen" Vereinbarung dem Variablennamen ein * vorangestellt wird.
  - **int *x, *y, z;**
  - **x** und **y** sind "Zeiger auf" int-Werte, **z** kann einen int-Wert aufnehmen.

- Der Name des Zeigers steht für den Zeiger (die Adresse) selbst.
- Um auf den Wert zuzugreifen, muss die Adresse dereferenziert werden. Dazu wird dem Namen des Zeigers (oder einem Ausdruck, der einen Zeiger liefert) ein * vorangestellt.

```
x=y;  //x zeigt nun auch dorthin, wo y hinzeigt
*x=5; //5 wird dort als Wert gespeichert, wo x hinzeigt
z=*y; //z bekommt den (int) Wert, auf den y zeigt (5)
x=5; x=z; //Indirektionslevel verschieden, keine implizite Typumwandlung
double *dp = {x} //Datentyp, auf den gezeigt wird, verschieden; keine implizite
                 //Typumwandlung
double d {*x}; //OK (implizite Typumwandlung)
```

universität wien

Zeiger (4)

• Um die Adresse einer Variablen zu ermitteln, wird dem Namen der Variable der Adressoperator & vorangestellt.

```
int *x, z {3};
x = &z;         //OK, &z ergibt eine Adresse; x zeigt nun auf z
*x = 7;         //Änderung des Werts, auf den x zeigt
cout << z;      //Ausgabe: 7
cout << *x+1; //Ausgabe: 8
cout << x;      //Ausgabe der (in x enthaltenen) Adresse von z
cout << &z;     //Ausgabe der gleichen Adresse wie oben
cout << &x;     //Ausgabe der Adresse von x (nicht wie oben)
cout << sizeof(x) //Ausgabe: 8 (auf 64bit Rechnern)
```

universität wien

Zeiger (5)

• Prinzipiell kann ein Zeiger auf jeden beliebigen Datentyp zeigen. Bei Zuweisungen zwischen Zeigern ist auf gleichen Datentyp zu achten, es werden keine impliziten Typumwandlungen durchgeführt.

```
int i {3}; double d {1.5}; int *ip {&i}; double *dp {&d};
*ip = *ip + *dp;  //OK, implizite Typumwandlungen; i wird 4
ip = dp;          //keine implizite Typumwandlung!
ip = (int *)dp;   //Erlaubt; (wahrscheinlich) überraschende
                  //Effekte
cout << *ip;      //Ausgabe: ?
```

universität wien

Zeiger (6)

- Zeiger können auch andere Zeiger referenzieren.

```
int i {3};
int *pi {&i};
int **ppi {&pi};
**ppi = *pi * 2;
```

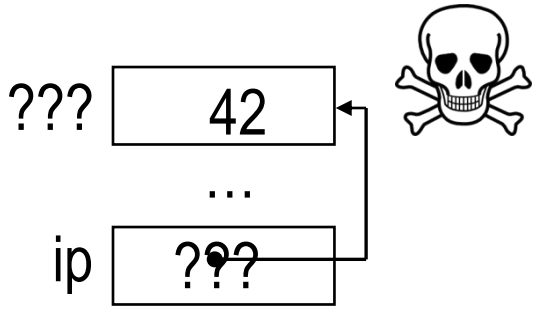
Keine Dereferenz-operatoren!

- Der Indirektionslevel auf beiden Seiten der Zuweisung muss gleich sein. Dabei gilt: Der Adressoperator & (der nur einmal auf eine Variable angewendet werden kann) erhöht den Level um eins und der Dereferenz-operator * (der auch mehrfach angewendet werden darf) vermindert den Level um eins.

Zeiger sind gefährlich

- Zugriff auf beliebige Speicherzellen möglich. (Wird von modernen Betriebssystemen meist ein wenig entschärft.)

```
int *ip;
*ip = 42;
```



- Defensive Programmierung: Zeiger immer initialisieren. Der Wert **`nullptr`** (früher auch 0) kann für Zeiger verwendet werden, die (noch) nirgendwohin zeigen. Beim Versuch den Nullzeiger (Nullpointer) zu dereferenzieren, wird das Programm abgebrochen. Es kann daher kein unerwünschter Speicherzugriff erfolgen.

```
int *ip {nullptr};
```

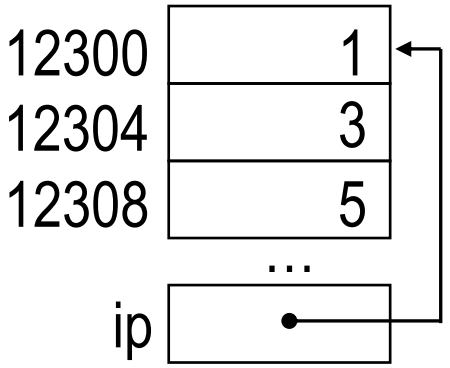
Laut C++ -Standard führt das Dereferenzieren des Nullpointers zu undefiniertem Verhalten (undefined behavior).
Auf den meisten aktuellen Systemen ist es aber unbedenklich

universität wien

Zeiger und Felder (1)

- Zu Zeigern (Adressen) können ganzzahlige Werte addiert werden.
- Für diese Addition wird die sogenannte Zeigerarithmetik verwendet:
- Der zum Zeiger addierte Wert wird als Offset in einem Feld interpretiert. Addition des Werts 1 ergibt somit nicht die nächste (Byte)Adresse, sondern die Adresse des nächsten Feldelements (unter der Annahme, dass der Zeiger ursprünglich auf ein Element eines Feldes zeigte).

```
int ia[] {1,3,5};
int *ip  {&ia[1] + 1};
cout << *ip;        //Ausgabe: 5
ip = ip - 2;
cout << *ip;        //Ausgabe: 1
cout << *(ip + 1); //Ausgabe: 3
cout << *ip + 1;   //Ausgabe: 2
```



ip ≡ 12300
ip+1 ≡ 12304 ≡ 12300+sizeof(*ip) ≡ 12300 + sizeof(int)

universität wien

Zeiger und Felder (2)

- Die Zeigerarithmethik erlaubt den Zugriff auf Feldelemente mittels eines Zeigers. Umgekehrt kann ein Array als konstanter Zeiger auf das erste Feldelement interpretiert werden.
- Zeigerarithmetik ist nur erlaubt, wenn der Zeiger und das Resultat der Operation in ein Array oder auf die Position direkt nach dem Ende des Array verweisen. Ansonst ergibt sich undefiniertes Verhalten.

- Es gilt: x[i] entspricht *(x+i), falls x ein Zeiger oder ein Array ist.
- Da ein Array technisch als Zeiger realisiert ist, ist es nicht notwendig (aber auch nicht verboten) den Adressoperator zu verwenden, um die Adresse des ersten Feldelements zu erhalten:

```
int ia[] {1,3,5};
int *ip1 {ia};       //gleicher Wert wie &ia (allerdings anderer Datentyp)
int *ip2 {ia+2};     //äquivalent zu &ia[2]
int *ip3 {ia[1]};    //Falsch!
int diff {ip2-ip1}; //diff == 2
char str[] {"C++"};
char *cp {str+1};
cout << cp;          //Ausgabe: ++ (I/O nur für char *)
cout << ip1;         //Ausgabe: Adresse von ia
```

Die Differenz zweier Zeiger (vom gleichen Typ) ergibt die Anzahl der Arrayelemente zwischen den beiden Adressen

Zeiger und Felder (3)

- Zeiger und Felder sind in C++ sehr ähnlich. Entscheidende Unterschiede sind:
  - Für Feldelemente wird automatisch Speicherplatz reserviert – ein Zeiger muss aber geeignet initialisiert werden:

```
int ia[] {1,3,5};
ia[1] = 10;
*ia = 0;
```

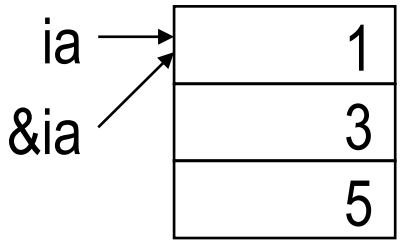
```
int *ip;
ip[1] = 10;
*ip = 0;
```

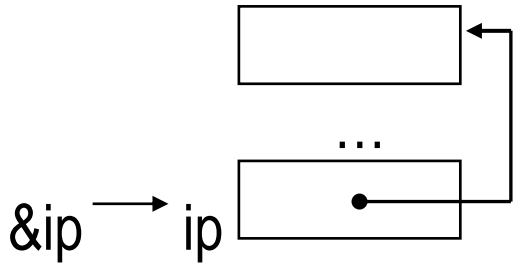
  - Das Feld ist ein konstanter Zeiger

```
ia = ip;
```

```
ip = ia;
```

  - Der Adressoperator liefert beim Feld einen Zeiger auf das erste Element, beim Zeiger wird der Indirektionslevel erhöht.





  - sizeof liefert bei einem Feld die Feldgröße, bei einem Zeiger die Größe eines Zeigers

universität wien

"Vergleich von Arrays"

- Zwei Arrays (z.B. Zeichenkettenkonstanten) können verglichen werden

```
char a[] {"C++"};
char b[] {"C++"};
assert(a==b); //schlägt fehl
```

- Es werden aber die Anfangsadressen (Zeiger auf die ersten Elemente verglichen)

- **Um die Inhalte zu vergleichen, muss eine Schleife verwendet werden!** (Für Zeichenketten stehen Funktionen wie strcmp zur Verfügung)

universität wien

Zeigerarithmetik und spezielle Operatoren

• Die Zuweisungsoperatoren **+=** und **-=** , sowie Inkrement- und Dekrementoperatoren (**++** und **--**) können auch im Zusammenhang mit Zeigern verwendet werden. Es gilt dann die übliche Zeigerarithmetik.

```
int arr[] {1,3,5};
int *ip {arr};
cout << *ip++;   //Ausgabe: 1 (++ bindet stärker)
cout << (*ip)++; //Ausgabe: 3
cout << *ip;     //Ausgabe: 4
cout << arr[1];  //Ausgabe: 4
```

• Achtung: Beziehungen wie **x[y]** entspricht ***(x+y)** und **++x** entspricht **x+=1** entspricht **x=x+1**, etc. gelten nur für die C++ Standardoperatoren, nicht aber für überladene Operatoren.

• Muss bei Bedarf extra programmiert werden!

universität wien

Zeiger, const und Überladen

- Es kann der Wert, auf den gezeigt wird, konstant sein

  `const T*`

- Es kann der Zeiger (die Adresse) selbst konstant sein

  `T *const`

- Und es kann beides konstant sein

  `const T *const`

- **`T*`** und **`const T*`** können überladen werden
- nicht aber **`T*`** und **`T *const`**

universität wien

void Zeiger

- **`void*`** ist ein Zeigertyp, der mit allen anderen Zeigertypen (mittels impliziter Typumwandlung) verträglich ist.
- Es ist garantiert, dass ein Zeiger, der in **`void*`** umgewandelt wird, bei der Rückumwandlung wieder den ursprünglichen Wert erhält.
- Das erlaubt, den Umgang mit Pointern auf unbekannte Datentypen
- Wird z. B. verwendet für die Kommunikation mit (älteren) Libraries und mit C-Interfaces

universität wien

Globale Funktionen begin() und end()

- Um die Anwendung von range-based-for-loops (for-each-Schleifen) auch für Arrays zu ermöglichen, wurden globale Funktionen **`begin()`** und **`end()`** definiert.

- Diese liefern einen Pointer auf das erste bzw. auf den virtuellen Eintrag hinter dem letzten Element des Arrays.
- Pointer (auf Arrayelemente) erfüllen alle Eigenschaften, die Iteratoren erfüllen müssen.

```
int a[10];
for (auto& i : a) i=0;  //Ginge das auch einfacher?
```

universität wien

Referenzen (Wiederholung)

- Referenzen ermöglichen es, einer Variablen einen zusätzlichen Namen zu geben. Eine Referenz wird mittels des Zeichens & definiert und muss initialisiert werden:

```
int i;
int &ir {i};
ir = 7;
cout << i; //Ausgabe: 7
```

  - Technisch ist eine Referenz ein Zeiger, der automatisch dereferenziert wird

- Die Adresse einer Referenz entspricht der Adresse der referenzierten Variable:

```
&i == &ir; //true
```

- Verwendung: Zur Vergabe von Namen für sonst unbenannte Variable; Zur Parameterübergabe bei Funktionsaufrufen, ...

```
int ia[] {1,3,5};
int &elem {ia[1]};
cout << elem; //Ausgabe: 3
```

universität wien

Kontextabhängige Interpretation der Operatoren * und &

- Die Operatoren * und & haben in C++ drei unterschiedliche Bedeutungen:
  - Multiplikation / bitweises Und:

```
i * j;
i & j;
```

  - Dereferenzieren / Adressoperator:

```
*ip = 5;
ip = &i;
```

  - Definition eines Zeigers / einer Referenz:

```
int *ip;
int &ir{i};
```

- Die genaue Bedeutung ist jeweils auf Grund des Kontexts ersichtlich.

```
int &ir1 {i = 3 * *(&i+2*i)};
```

universität wien

Zeiger als Funktionsparameter

- Pointer eröffnen eine weitere Möglichkeit, um Referenzparameter zu realisieren.

```
void sort(int* i, int* j) {
  if (*i>*j) {
    int help {*i};
    *i = *j;
    *j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(&a,&b);
  assert(a<=b);
  return 0;
}
```

Ist nur lesender Zugriff erforderlich, kann hier **`const int*`** verwendet werden.

Komplexere Syntax für den Zugriff

Beim Aufruf als Referenzparameter identifizierbar

universität wien

Felder als Funktionsparameter

• Arrays werden immer als Zeiger übergeben und niemals kopiert. Daher sind folgende Funktionsdeklarationen äquivalent:

```
void foo(int par1[5]);
void foo(int par1[]);
void foo(int *par1);
```

• Bei mehrdimensionalen Arrays werden die Elementanzahlen für die Adressberechnung benötigt. Es darf daher nur die erste („innerste“) Dimension unterdrückt werden:

```
void foo(int par1[][4][5]);
```

Bei Verwendung eines Zeigers als Parameter muss die Adressberechnung „händisch“ programmiert werden.

universität wien

Dynamischer Speicher

• Wird während des Programmlaufs zusätzlich zu den vorhandenen Variablen Speicherplatz benötigt, so kann dieser vom Betriebssystem angefordert werden. Der Operator new liefert einen Zeiger auf ein Element vom gewünschten Typ:

```
int *ip  {new int};
*ip = 5;
int &ir {*ip};
cout << ir; //Ausgabe: 5
```

Ist kein Speicherplatz mehr vorhanden, dann löst new eine Exception vom Typ std::bad_alloc aus. (Wahlweise kann auch der Nullzeiger zurückgeliefert werden.)

• Der allozierte Speicher bleibt so lange reserviert, bis er mit dem Operator delete wieder freigegeben (oder das Programm beendet) wird.

```
delete ip;
*ip = 5;     //Effekt nicht definiert, da Speicher nicht mehr reserviert
ip = nullptr;  //Nach delete Zeiger auf nullptr setzen (defensive Programmierung)
```

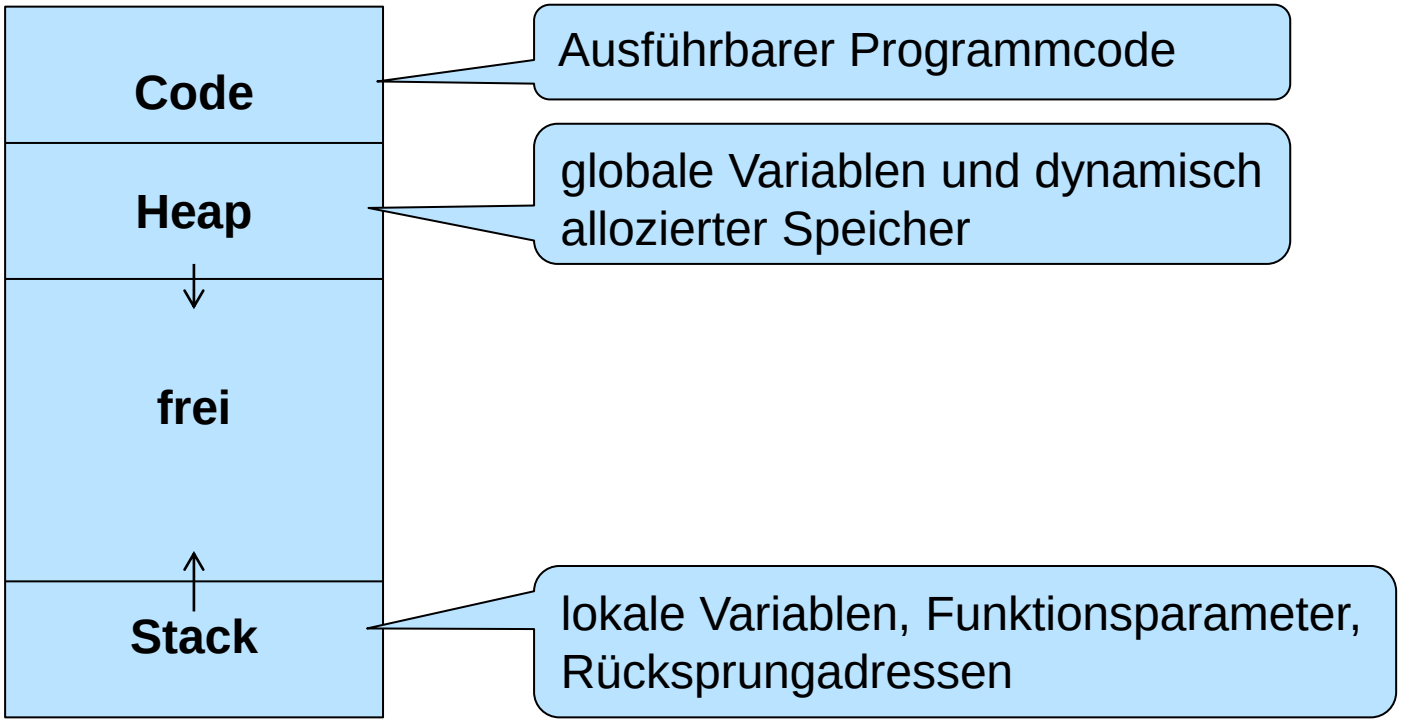
• Mit dem Operator new[ ] kann ein ganzes Feld alloziert werden. Dieses muss dann mittels delete[ ] wieder freigegeben werden.

```
ip = new int[10];
ip[5] = 5;
delete[] ip;
```

Position der Klammern beachten!

universität wien

Memory Layout (prinzipiell)



universität wien

Initialisierung von dynamisch alloziertem Speicher

- C++ hat lange keine syntaktische Möglichkeit angeboten, dynamisch allozierten Speicher direkt zu initialisieren.

- Seit C++98 Möglichkeit der „zero initialization“: new int[10]()

- Mit C++03 uminterpretiert zu „value initialization“: Konstruktoraufruf, falls Konstruktor definiert, sonst mit 0

- Seit C++11 neue durchgängige Initialisierungssyntax („Universal Initalizer“): new int[]{1,2}

Verfügbarkeit abhängig von der verwendeten Compilerversion. Eventuell müssen Optionen beim Start des Compilers gesetzt werden.

- Kleinere Änderungen noch mit C++14

universität wien

new und delete

• Um Speicherlecks (memory leaks) zu vermeiden, ist darauf zu achten, dass zu jedem **`new`** ein passendes **`delete`** Statement ausgeführt wird.

```
int *p1, *p2, *p3;
p1 = new int;
p2 = p1;
p3 = new int[10];
delete p2;      //OK
delete p1;      //Fehler! Schon freigegeben, undefiniert
delete p3;      //Fehler! Falsches delete, undefiniert
p2 = nullptr;
delete p2;      //OK (kein Effekt)
delete[] p2;    //OK (kein Effekt)
```

universität wien

Verwendung von Arrays und Pointern

- Arrays und Pointer bieten die Möglichkeit hardwarenahe zu programmieren.

- Für die meisten Aufgabenstellungen stehen bessere Konstrukte (z.B. std::vector bzw. std::array) zur Verfügung. Diese sollten in der Regel verwendet werden.

- Notwendig sind Arrays und Pointer, um Klassen wie z.B. std::vector zu implementieren, zur Vermeidung von Kopien, zur Performanceoptimierung (sehr selten notwendig), zur Kommunikation mit anderen Programmiersprachen oder Verwendung von (älteren) Libraries.

- Pointer werden darüber hinaus für **`this`** und für die Implementierung von rekursiven Datenstrukturen verwendet
  z. B.: **`class List { int element; List *next;}`**)

universität wien

this

- Innerhalb einer Methode ist **`this`** ein Pointer auf das Objekt, für welches die Methode aufgerufen wurde.

- Der Datentyp von this für eine Klasse C ist **`C *const`** (konstanter Pointer auf ein Objekt vom Typ C) bzw. **`const C *const`** (konstanter Pointer auf ein konstantes Objekt vom Typ C) für const-Methoden)

- *this ist damit das Objekt, für welches die Funktion aufgerufen wurde.

- Der Operator **`->`** ist eine Kurzschreibweise für Member-Zugriff über einen Pointer:

  `x->y` ⟺ **`(*x).y`** (die Klammen sind hier notwendig, da **`.`** stärker bindet als `*`)

universität wien

Beispiel: Zeigerverbiegen (1)

```
#include<iostream>
using namespace std;
int main() {
  cout << "Zeigerverbiegungen\n\n";
  int *ip;
  { // warum wohl diese Klammern ??
    cout << "Der Tragoedie erster Teil: Datentyp int\n";
    int v {4711};
    int *pv {&v};
    int **ppv {&pv};
    cout << "Die Werte sind: " << v << ", " << (void *)pv << " und " << (void *)ppv << '\n';
    cout << "v = " << v << " = " << *pv << " = " << **ppv << '\n';
    int f[10]{0,1,2,3,4,5,6,7,8,9};
    for (int i{0}; i < 10; i=i+1)
      cout << "Index: " << i << " Wert: " << f[i] << " Adresse: " << (void *)(f + i) << '\n';
    pv = f + 9; // Aequivalent zu &f[9]
    cout << "Letzter Wert indirekt: " << *pv << '\n';
    ip = new int; // warum nicht ip = &v?
    *ip = v;
  }
  {
    cout << "\nFunktioniert aber auch mit double\n";
    double v {47.11};
    double *pv {&v};
    double **ppv {&pv};
    cout << "Die Werte sind: " << v << ", " << (void *)pv << " und " << (void *)ppv << '\n';
    cout << "v = " << v << " = " << *pv << " = " << **ppv << '\n';
    double f[10]{0,0.5,1,1.5,2,2.5,3,3.5,4,4.5};
    for (int i{0}; i < 10; i=i+1)
      cout << "Index: " << i << " Wert: " << f[i] << " Adresse: " << (void *)(f + i) << '\n';
    pv = f + 9; // Aequivalent zu &f[9]
    cout << "Letzter Wert indirekt: " << *pv << '\n';
    cout << "Dynamisches Element: " << *ip << '\n';
  }
```

universität wien

Beispiel: Zeigerverbiegen (2)

```
  {
    cout << "\nUnd erst recht mit char\n";
    char v {'x'};
    char *pv {&v};
    char **ppv {&pv};
    cout << "Die Werte sind: " << v << ", " << (void *)pv << " und " << (void *)ppv << '\n';
    cout << "v = " << v << " = " << *pv << " = " << **ppv << '\n';
    char f[10]{'z','a','y','b','x','c','w','d','v','e'};
    for (int i{0}; i < 10; i=i+1)
      cout << "Index: " << i << " Wert: " << f[i] << " Adresse: " << (void *)(f + i) << '\n';
    pv = f + 9; // Aequivalent zu &f[9]
    cout << "Letzter Wert indirekt: " << *pv << '\n';
    delete ip;
  }
  cout << "\n\n";
  return(0);
}
```

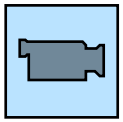
Beispiel: Zeigerverbiegen (Ausgabe)

```
Zeigerverbiegungen

Der Tragoedie erster Teil: Datentyp int
Die Werte sind: 4711, 0x7fff232dbdbc und 0x7fff232dbda8
v = 4711 = 4711 = 4711
Index: 0 Wert: 0 Adresse: 0x7fff232dbd80
Index: 1 Wert: 1 Adresse: 0x7fff232dbd84
Index: 2 Wert: 2 Adresse: 0x7fff232dbd88
Index: 3 Wert: 3 Adresse: 0x7fff232dbd8c
Index: 4 Wert: 4 Adresse: 0x7fff232dbd90
Index: 5 Wert: 5 Adresse: 0x7fff232dbd94
Index: 6 Wert: 6 Adresse: 0x7fff232dbd98
Index: 7 Wert: 7 Adresse: 0x7fff232dbd9c
Index: 8 Wert: 8 Adresse: 0x7fff232dbda0
Index: 9 Wert: 9 Adresse: 0x7fff232dbda4
Letzter Wert indirekt: 9

Funktioniert aber auch mit double
Die Werte sind: 47.11, 0x7fff232dbdb0 und
x7fff232dbda8
v = 47.11 = 47.11 = 47.11
Index: 0 Wert: 0 Adresse: 0x7fff232dbd30
Index: 1 Wert: 0.5 Adresse: 0x7fff232dbd38
Index: 2 Wert: 1 Adresse: 0x7fff232dbd40
Index: 3 Wert: 1.5 Adresse: 0x7fff232dbd48
Index: 4 Wert: 2 Adresse: 0x7fff232dbd50
Index: 5 Wert: 2.5 Adresse: 0x7fff232dbd58
Index: 6 Wert: 3 Adresse: 0x7fff232dbd60
Index: 7 Wert: 3.5 Adresse: 0x7fff232dbd68
Index: 8 Wert: 4 Adresse: 0x7fff232dbd70
Index: 9 Wert: 4.5 Adresse: 0x7fff232dbd78
Letzter Wert indirekt: 4.5
Dynamisches Element: 4711

Und erst recht mit char
Die Werte sind: x, 0x7fff232dbdbc und 0x7fff232dbda8
v = x = x = x
Index: 0 Wert: z Adresse: 0x7fff232dbdc0
Index: 1 Wert: a Adresse: 0x7fff232dbdc1
Index: 2 Wert: y Adresse: 0x7fff232dbdc2
Index: 3 Wert: b Adresse: 0x7fff232dbdc3
Index: 4 Wert: x Adresse: 0x7fff232dbdc4
Index: 5 Wert: c Adresse: 0x7fff232dbdc5
Index: 6 Wert: w Adresse: 0x7fff232dbdc6
Index: 7 Wert: d Adresse: 0x7fff232dbdc7
Index: 8 Wert: v Adresse: 0x7fff232dbdc8
Index: 9 Wert: e Adresse: 0x7fff232dbdc9
Letzter Wert indirekt: e
```



universität wien

Zeigerverbiegen die zweite

```
#include<iostream>
using namespace std;
void f (int i1, int *pi2, int &ri3) {
  i1--;
  *pi2++;
  ri3 += 5;
}
void f1 (int i1, int *pi2, int &ri3){
  i1--;
  (*pi2)++; // Auf die Klammer kommt es an
  ri3 += 5;
}
void f2 (int **pp, int *p) {
  int local {*p};
  *pp = p;
  // Was wuerde passieren, wenn man
  // stattdessen *pp = &local; verwendete
}
int main() {
  cout << "Der Tragoedie zweiter Teil\n";
  int v1 {4711}, v2 {815}, v3 {666};
  int *pv2 {&v2};
  int *pv3 {&v3};
  int &rv2 {v2};
  int **ppv2 {&pv2};
  f(v1, &rv2, v3);
  cout << v1 << ", " << v2 << ", " << v3 << '\n';
  f(v1, *ppv2, *pv3);
  cout << v1 << ", " << v2 << ", " << v3 << '\n';
  f1(v1, *ppv2, *pv3);
  cout << v1 << ", " << v2 << ", " << v3 << '\n';
  f2(ppv2, pv3);
  cout << v2 << ", " << **ppv2 << '\n';
  cout << "\n\n";
  return(0);
}
```

```
Der Tragoedie zweiter Teil
4711, 815, 671
4711, 815, 676
4711, 816, 681
816, 681
```

universität wien

Wiederholung

- Feldvereinbarung
- Feldinitialisierung
- Feldverwendung
- Zeichenketten
- Zeiger
- Zeigerarithmetik
- Referenzen
- Zeiger als Parameter
- Arrays als Parameter
- Dynamischer Speicher

```
int x[10]; int y[2][2];
int x[10] {7, 4};
char s[] {"C++"};
char s[] {'C','+','+','\0'};
x[0] = x[9] - x[8];
char s[] {"C++"};
cout << sizeof(s) << sizeof(s[0]);
int i, *p; p = &i; *p = i+1;
char s[] {"C++"}; char *p = s+2;
int i, &ri {i};
void f(int *ptr);
void f(int arr[]);
char *s {new char}; delete s;
char *s {new char[7]}; delete[] s;
```

universität
wien

14. Klassen revisited

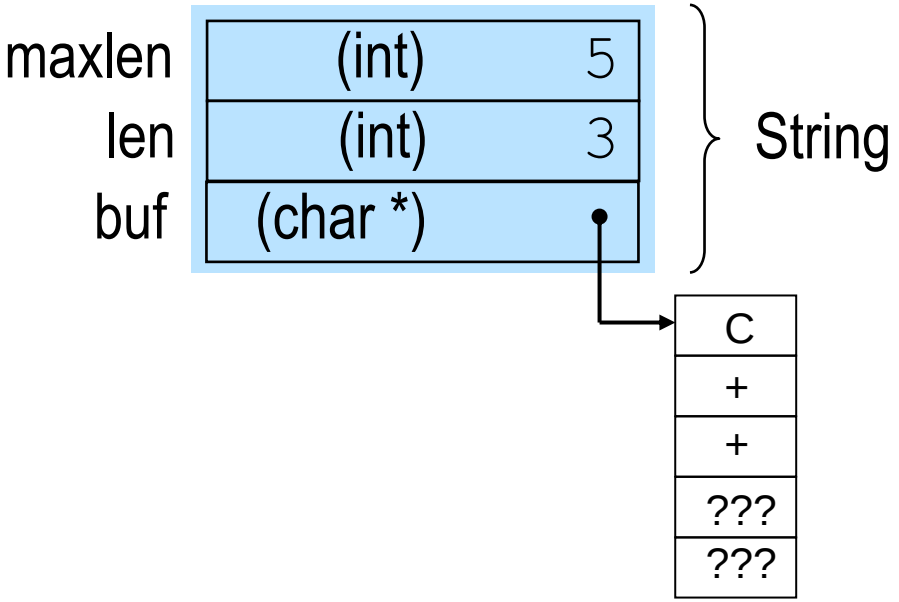
universität wien

Klasse String (1)

- Im Folgenden gehen wir von der Implementierung einer simplen Klasse String aus, die das zur Speicherung erforderliche Memory dynamisch verwaltet.

Klasse String (2)

- Ein String
  - besteht aus
    - –einer **`int`**-Zahl für die maximale Länge und
    - –einer **`int`**-Zahl für die aktuelle Länge und
    - –einem **`char`**-Array für den Inhalt (die gespeicherten Zeichen)
  - und man kann mit ihm einiges machen:
    - –einen anderen String anhängen,
    - –einen Teilstring suchen/ersetzen,
    - –Zugriff auf eine bestimmte Zeichenposition
    - –Ausgeben/Einlesen
    - –…



universität wien

String (3)

```
class String {
  int maxlen, len;
  char *buf;
public:
  String(int maxlen = 0, const char* cstr = nullptr);
  String(const char* cstr);
  void set (const char* source);
  int find(const String& substr);
  String operator+ (const String& rightop);
  String operator- (const String& rightop);
  char& operator[](int index);
  void print ();
};
String fill (char c, int count);
```

universität wien

String (4)

```
String::String(int maxlen, const char* cstr) {
  if (maxlen<0) maxlen = 0;
  this -> maxlen = maxlen;
  len = cstr ? strlen(cstr) : 0;
  if (len>maxlen) len = maxlen;
  buf = maxlen ? new char[maxlen] : nullptr;
  for (int i=0; i<len; ++i) buf[i] = cstr[i];
}

String::String(const char* cstr) {
  maxlen = len = strlen(cstr);
  buf = maxlen ? new char[maxlen] : nullptr;
  for (int i=0; i<len; ++i) buf[i] = cstr[i];
}
```

universität wien

Probleme mit der Klasse String

- Die Klasse String ist in dieser Implementierung leider nicht verwendbar.

Bei der praktischen Anwendung treten unterschiedliche Probleme auf. Zum Beispiel:

```
for (int i {0}; i>=0; ++i) {String s(100000); cout << i << '\n';}
```

Ausgabe:

```
0
1
…
32040
32041
terminate called after throwing an instance of 'std::bad_alloc'
  what():  St9bad_alloc
Aborted
```

Warum?

universität wien

Destruktoren (1)

- Wird ein String-Objekt zerstört, so geht der für buf reservierte Speicher „verloren“, da kein Aufruf des delete[ ]-Operators erfolgt.

- Abhilfe: Destruktor-Methode
  - kann als Gegenstück zum Konstruktor aufgefasst werden
  - dient „zum Aufräumen“ (statt zum Initialisieren)
  - heißt wie die Klasse, allerdings mit vorangestelltem ~ (~String)
  - wird automatisch aufgerufen, wenn ihr Objekt zu existieren aufhört
  - darf über keine Parameter verfügen (daher nicht überladbar)
  - hat keinen Rückgabewert

```
class String {
//alles wie bisher
…
public:
…
   ~String() {if (buf) delete[] buf;}
};
```

Im Sinne einer defensiven Programmierung könnten **buf, len** und **maxlen** wieder auf **nullptr**, bzw 0 gesetzt werden. Ist aber nicht notwendig und ineffizient.

universität wien

Destruktoren (2)

• Der Destruktor gibt nun den allozierten Speicherplatz wieder ordnungsgemäß frei, wenn ein Objekt zerstört wird (entweder automatisch oder mit delete bzw. delete[ ]).

```
for (int i {0}; i>=0; ++i) {String s(100000); cout << i << '\n';}
```

Ausgabe:

```
0
1
…
2147483646
2147483647
```

Schleife wird normal beendet. (Sollte das nicht eine Endlosschleife sein?)

universität wien

Destruktoren (3)

- Ein Objekt gilt als zerstört, sobald der Destruktor zu laufen beginnt. Die Lebensdauer eines Objekts ist somit vom Ende (der schließenden Klammer) des Konstruktors bis zum Anfang (zur öffnenden Klammer) des Destruktors.

- Wird kein Destruktor definiert, so wird einer automatisch vom Compiler erzeugt. Dieser kümmert sich nur um die Freigabe der automatisch erzeugten Instanzvariablen.

- Ein Destruktor sollte keine Exception werfen (wird während der Abarbeitung einer Exception (stack unwinding) durch einen Destruktor eine weitere Exception geworfen, so wird das Programm abgebrochen). Verwendet der Destruktor Aufrufe, die eventuell eine Exception werfen könnten, so empfiehlt es sich, diese zu fangen.

```
~Klasse() noexcept {
  try{…}
  catch(...){…}
}
```

universität wien

Destruktoren (4)

LIFO
Warum?

- Für automatische Objekte werden die Destruktoren in genau der umgekehrten Reihenfolge aufgerufen, wie die Konstruktoren!

```
class KDTest {
  static int no;
  int val;
public:
  KDTest() {cout<<"K"<<(val=no++)<<'\n';}
  ~KDTest() {cout<<"D"<<val<<'\n';}
};

KDTest::no {0};

class A {
  KDTest kda;
};

class B {
  KDTest kd1;
  KDTest kd2;
};

void f() {
  KDTest kd;
  B b;
}
```

```
main() {
  KDTest kd;
  f();
  B b;
  for (int i{0};i<2;++i) KDTest kd;
  A kd1;
}
```

```
K0
K1
K2
K3
D3
D2
D1
K4
K5
K6
D6
K7
D7
K8
D8
D5
D4
D0
```

universität wien

Weitere Probleme mit Klasse String

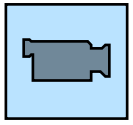
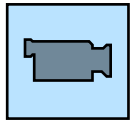
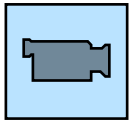
```
main() {
  String s {"Init"};
  String s1 {s};
}
```

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

```
main() {
  String s {"Init"};
  (s-"ni").print();
}
```

```
*** glibc detected *** double free or corruption (fasttop): 0x0804c120 ***
Aborted
```

universität wien

Kopierkonstruktoren (1)

- Problemlösung: Kopierkonstruktor-Methode
  - Konstruktor mit Argument vom selben Datentyp:

```
String (const String& original);
```

  - Argument muss (i.a. const-) Referenz sein! (Warum?)
  - wird aufgerufen, wenn ein Objekt mit einem anderen initialisiert wird

```
String t {s}; // oder String t = s oder String t(s)
              // auch bei call by value und bei temporären
              // Variablen als Retourwert von Funktionen

String::String (const String& original) {
  maxlen = original.maxlen; len = original.len;
  buf = maxlen ? new char[maxlen] : nullptr;
  for (int i=0; i<len; ++i) buf[i] = original.buf[i];
}
```

universität wien

Kopierkonstruktoren (2)

- Wird kein Kopierkonstruktor definiert, aber einer benötigt (weil Objekte kopiert werden müssen), so wird vom Compiler einer automatisch erzeugt!

- Ein vom Compiler erzeugter Kopierkonstruktor kopiert einfach die Instanzvariablen der Reihe nach = komponentenweise oder flache Kopie (bei Objekten mit dynamischem Speicher fast immer unerwünscht !)

```
class Point {
  String label;
  double x;
  double y;
  …
};
```

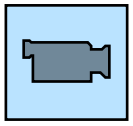
Automatisch erstellter Kopierkonstruktor und Destruktor würden ausreichen.

universität wien

Noch immer nicht alle Probleme der Klasse String gelöst

```
main() {
  String s {"Init"};
  String s1;
  s1 = s;
}
```

```
*** glibc detected *** double free or corruption (fasttop): 0x0804c120 ***
Aborted
```



universität wien

Zuweisungsoperatoren (1)

- Problemlösung: Zuweisungsoperator-Methode
  - **`String& operator= (const String& rightop);`**
  - Argument **sollte** (muss aber nicht) (wieder i.a. const-) **Referenz** sein
  - wird bei Zuweisungen mit passender rechter Seite aufgerufen
  - kann u.U. **überladen** sein

```
String& operator= (const char* cstr);
String& operator= (const int val);
```

Aufgrund der impliziten Typumwandlung nicht notwendig

  - gibt **konventionsgemäß** (aber nicht unbedingt) eine **Referenz auf die linke Seite** der Zuweisung zurück (das ermöglicht Verkettungen von Zuweisungen in der Art **`a = b = c`**)

Zuweisungsoperatoren (2)

```
String& String::operator= (const String& rightop) {
  if (&rightop == this) return *this;
  if (buf) delete[] buf;
  maxlen = rightop.maxlen; len = rightop.len;
  buf = maxlen ? new char[maxlen] : nullptr;
  for (int i {0}; i<len; ++i) buf[i] = rightop.buf[i];
  return *this;
}
```

Verhindert Fehler bei Zuweisung einer Variable an sich selbst

Eventuell bereits reservierten Puffer wieder freigeben (Wiederverwendung auch möglich, aber verkompliziert meist den Code)

Rest wie in Kopierkonstruktor

Retourwert für verkettete Zuweisungen

- Wird kein Zuweisungsoperator definiert, aber einer benötigt (weil Objekte kopiert werden müssen), so wird vom Compiler einer automatisch erzeugt (nur für Zuweisungen innerhalb der gleichen Klasse)!
- Ein vom Compiler erzeugter Zuweisungsoperator führt eine komponentenweise Zuweisung der Instanzvariablen durch (bei Objekten mit dynamischem Speicher fast immer unerwünscht !

universität wien

Zuweisungsoperatoren (3) Exception Safety

```
String& String::operator= (const String& rightop) {
  //if (&rightop == this) return *this; nicht mehr notwendig
  char *nbuf = rightop.maxlen ? new char[rightop.maxlen] : nullptr;
  for (int i {0}; i<rightop.len; ++i) nbuf[i] = rightop.buf[i];
  if (buf) delete[] buf;
  buf = nbuf;
  maxlen = rightop.maxlen;
  len = rightop.len;
  return *this;
}
```

- Exception Safety bedeutet, dass Methoden so formuliert werden, dass gewisse Garantien gegeben werden können
  - no-throw guarantee: Es gibt keinesfalls eine Exception; Aufruf ist immer erfolgreich
  - strong guarantee: Programmzustand vor dem Methodenaufruf bleibt im Fall einer Exception
  - basic guarantee: Objekt bleibt in einem konsistenten Zustand; keine resource leaks
  - no guarantee:

universität wien

Zuweisungsoperatoren (4) copy and swap

- Alternativ kann der Zuweisungsoperator auch so implementiert werden

```
String& String::operator= (String rightop) {
  std::swap(maxlen, rightop.maxlen);
  std::swap(len, rightop.len);
  std::swap(buf, rightop.buf);
  return *this;
}
```

copy

swap
(oft in einer eigenen swap Methode der Klasse realisiert)

- Eine konzise und simple Vorgehensweise, allerdings nicht effizient, wenn der schon vorhandene Speicher wiederverwendet werden könnte.

universität wien

Zuweisungsoperatoren (4)

```
class Point {
  String label;
  double x;
  double y;
  …
};
```

Automatisch erstellter Zuweisungsoperator würde ausreichen.

universität wien

Zuweisung versus Initialisierung (1)

- Unterscheidung i.A. nur bei Vorhandensein dynamischer Datentypen relevant!
- Initialisierungen erfolgen bei neuen (“nackten”) Objekten
  - Objekt muss erst aufgebaut werden
  - Zuständig: Konstruktor(en), insb. auch Kopierkonstruktor
- Zuweisungen erfolgen auf bereits bestehende (“fertige”) Objekte
  - Alte Objektstruktur kann u.U. wiederverwertet werden
  - Falls nicht möglich: ordentlich zurückgeben, dann Objekt neu aufbauen, dann Daten der rechten Seite übernehmen
  - Zuständig: Zuweisungsoperator(en)
- Parameterübergabe: Initialisierung der formalen Parameter
- Retourwert: Initialisierung eines (temporären) Wertes

universität wien

Zuweisung versus Initialisierung (2)

- Konstruktoren sollen i.A. die Instanzvariablen ihrer Objekte initialisieren.
- Bisher jedoch durch Zuweisungen bewerkstelligt:

```
String::String() { maxlen = 0; len = 0; …}
```

- Sauberere Lösung durch eigene Initialisierungssyntax (nur bei Definition von Konstruktoren erlaubt):

```
String::String(): maxlen{0},len{0} { … }
String::String(int maxlen) : maxlen{maxlen}, len{0} { … }
```

Implizit korrekte Auflösung der beiden gleichbenannten Variablen

- Hier egal - bei Instanzvariablen mit Klassen-Datentypen (Aktivierung von Kopierkonstruktor oder Zuweisungsoperator!) und bei Instanzvariablen, die konstant (**`const`**) oder Referenzen (**`Typ&`**) sind, aber wichtig!

universität wien

Zuweisung versus Initialisierung (3)

```
class Point {
  String label;
  double x;
  double y;
  …
};
Point::Point(String label) {
  this->label = label;
}
```

```
Point::Point(String label):label{label} {
}
```

Aufruf des Zuweisungsoperators (**falsch**, wenn Instanzvariable **`label`** noch nicht initialisiert ist. Initialisiert der (eventuell vom Compiler automatisch erzeugte) Defaultkonstruktor für String die Instanzvariable **`label`** nicht, so wird eventuell versucht, den Zeiger **`label.buf`**, der uninitialisiert ist, freizugeben; jedenfalls **ineffizient**)

Aufruf des Kopierkonstruktors

universität wien

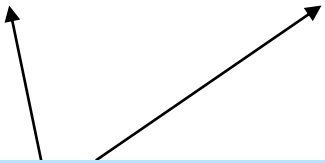
Zuweisung versus Initialisierung (4)

- Faustregel:

Instanzvariablen immer mit der Initialisierungssyntax initialisieren.

- Wichtiges Detail:

```
String::String() : len{++n}, maxlen{n} {}
```



Reihenfolge der Angabe hier ist **irrelevant**!
Wird in der **Reihenfolge der Deklaration**
der Instanzvariablen in der Klasse abgearbeitet!
(Der Compiler weist eventuell durch eine Warnung darauf hin.)

universität wien

Initialisierungssyntax

- Wir verwenden möglichst durchgehend die neuere Initialisierung mit geschwungenen Klammern (brace-init-list).
- Möglich ist es aber auch oft, runde Klammern zu verwenden und = (das in diesem Fall kein Zuweisungsoperator ist):

```
int i = 0;
String s1("abc"), s2 = "xyz"; //Keine Zuweisung!
String::String() : len(++n), maxlen(n){}
std::vector<int>{10} //Vektor mit einem Element 10
std::vector<int>(10) //Vektor mit 10 Elementen 0
```

- Wie die letzten beiden Beispiele illustrieren, muss manchmal noch die ältere Form angewendet werden, um den gewünschten Konstruktor zu wählen.

universität wien

Resource Acquisition is Initialization (RAII)

```
void foo() {
  try {
    char *buf {new char[10]};
  }
  catch(std::bad_alloc) {
    return; //kein delete!
  }
  try {
    …
    delete[] buf;
  }
  catch(...) {
    delete[] buf;
  }
}
```

Mühselig und fehleranfällig, vor allem, wenn mehrere Ressourcen verwendet werden

```
class BufClass {
  char *buf;
public:
  BufClass():buf {new char[10];} {}
  ~BufClass() {delete[] buf;}
  …
};
void foo() {
  BufClass b;
  …
}
```

Eventuell try-catch-Block anwenden, um Fehler bei der Erzeugung des Objekts abzufangen

Destruktor wird ganz sicher aufgerufen, wenn das Objekt zerstört wird (allerdings gibt es kein Objekt, wenn der Konstruktor fehlschlägt und damit auch keinen Destruktoraufruf)

Dieses Konzept ist für beliebige Ressourcen (Speicherplatz, Dateien, Drucker, Geräte, …) einsetzbar!

universität wien

Smart Pointers

- Zur Vereinfachung des Umgangs mit Ressourcen gibt es sogenannte smart pointer, die die Ressource auf die sie zeigen "besitzen" und automatisch dafür sorgen, dass die Ressource freigegeben wird, sobald der (letzte) Pointer auf die Ressource zerstört wird (out-of-scope geht).

- Die Standardlibrary bietet hier vor allem drei Typen von smart pointern an:

  `std::unique_ptr`, **`std::shared_ptr`** und **`std::weak_ptr`**.

universität wien

Wiederholung

- Destruktor `~String();`
- Kopierkonstruktor `String(const String&);`
- Zuweisungsoperator `String& operator=(const String&);`
- Zuweisung vs. Initialisierung `String s {"Init"}; s="Assign";`
- Initialisierungssyntax `String(int maxlen):maxlen(maxlen);`
- Resource Acquisition Is Initialization `SpeicherObjekt so;`

Für Klassen, in denen dynamischer Speicher verwendet wird, müssen in aller Regel ein Destruktor, ein Kopierkonstruktor und ein Zuweisungsoperator implementiert werden. ("rule of three" bzw. "the big three")

universität
wien

15 Vererbung

- Inheritance

universität wien

Vererbung

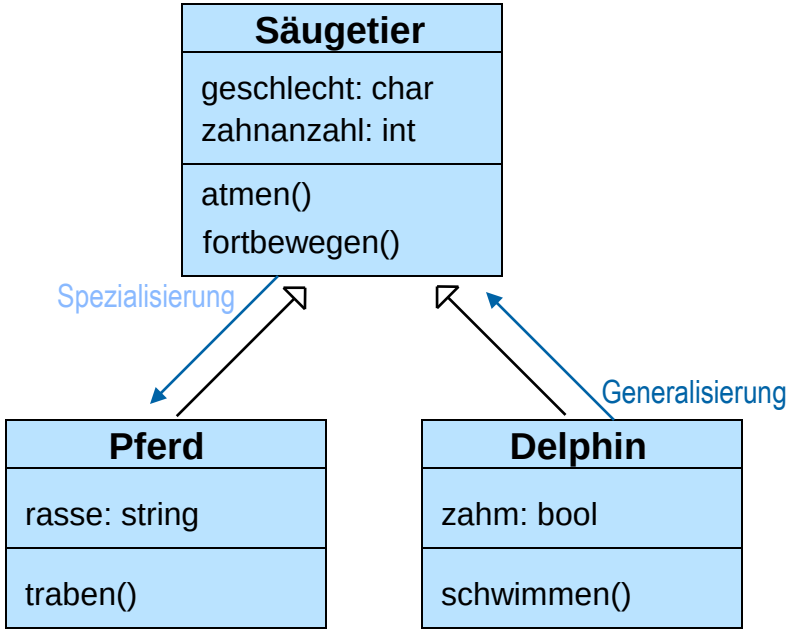
- Vererbung bildet mit Datenkapselung und Polymorphismus die drei grundlegenden Konzepte der Objektorientierung

- Sie erlaubt die einfache Wiederverwendung von bereits produziertem Code (code reuse) und damit
  - **Einfachere Programmierung**
  - **Einfacheres Fehlersuchen und Testen**

- Üblicherweise erbt eine spezialisierte Klasse (Unterklasse) von einer generelleren Klasse (Oberklasse, Basisklasse). Dabei gilt eine „is-a“ Beziehung (jedes Element der Unterklasse ist auch ein Element der Oberklasse)

Beispiel

- UML



- C++

```
class Saeugetier {
   char geschlecht;
   int zahnanzahl;
public:
   void atmen();
   void fortbewegen();
};

class Pferd: public Saeugetier {
   string rasse;
public:
   void traben();
};

class Delphin: public Saeugetier {
   bool zahm;
public:
   void schwimmen();
};
```

Ein Pferd ist eine spezielle Art Säugetier und hat damit alle Eigenschaften und Methoden der Klasse Säugetier, sowie zusätzlich diejenigen der Klasse Pferd (analog für Delphin)

universität wien

Substitutionsprinzip (Liskov)

• Ein Objekt der abgeleiteten (erbenden Klasse) kann an jeder beliebigen Stelle für ein Objekt der Basisklasse eingesetzt werden.

z. B.:

```
void f(Saeugetier);

Saeugetier s;
Pferd p;
s.atmen();  //OK
p.traben();//OK
f(p); //OK
s = p;      //OK
s.atmen();  //OK
s.traben();//nicht erlaubt
p = s;      //nicht erlaubt
```

universität wien

Vererbung ohne Substitutionsprinzip

- Die Mechanismen, die von Programmiersprachen zur Vererbung zur Verfügung gestellt werden, erlauben meist auch ein „Unterlaufen“ des Substitutionsprinzips.

- Das kann manchmal nutzbringend verwendet werden, führt aber bei Unvorsichtigkeit zu Problemen. Sollte daher nur eingesetzt werden, wenn wirklich notwendig.

z.B. Einschränkung:

```
class Untier: private Saeugetier {
};

Untier u;
u.atmen();          //nicht erlaubt (private Methode)
Saeugetier t{u}; //nun auch nicht mehr erlaubt
```

Private Vererbung: Alle Methoden und Instanzvariablen der Oberklasse sind in der Unterklasse private
Default Vererbung ist private für Klassen und public für Structs

universität wien

Überladen von Methoden der Basisklasse

• Oft ist es notwendig, in der erbenden Klasse das Verhalten der Basisklasse nicht komplett zu übernehmen, sondern zu adaptieren.

z.B.: Delphine müssen zum Atmen erst auftauchen.

```
class Delphin: public Saeugetier {
   bool zahm;
public:
   void schwimmen();
   void auftauchen();
   void atmen() {auftauchen(); …}
};

Saeugetier s;
Delphin d;
d.atmen(); //Delphin::atmen
s=d;
s.atmen(); //Saeugetier::atmen! Information, dass es sich um
           //einen Delphin handelt ist verloren gegangen
```

Override: Delphin::atmen überlagert alle (eventuell überladenen) Methoden atmen der Basisklasse (**`Saeugetier::atmen`**)
Dies kann bei Bedarf durch eine using-Deklaration rückgängig gemacht werden:
`using Sauegetier::atmen;`

universität wien

Aufruf der Methode der Basisklasse

• Wie in unserem Beispiel ist es sehr oft nützlich, die Funktionalität der Basisklasse in einer überlagerten Methode weiterhin zu nutzen.

```
class Delphin: public Saeugetier {
   bool zahm;
public:
   void schwimmen();
   void auftauchen();
   void atmen() {
     auftauchen();
     Saeugetier::atmen();
   }
};
```

Taucht zunächst auf und führt dann „normale“ Atmung aus

universität wien

Implizite Pointerkonversion

• Normalerweise findet zwischen Zeigern unterschiedlichen Typs keine implizite Typumwandlung statt:

```
int *x; double *y{x};
```

• Ein Zeiger (eine Referenz) auf eine abgeleitete Klasse kann aber implizit auf einen Zeiger (eine Referenz) auf eine Basisklasse umgewandelt werden:

```
Delphin d; Saeugetier *sp{&d}; Saeugetier &sr{d};
```

- Diese Art der Umwandlung wird (als einzige) auch bei catch-Statements implizit durchgeführt. Man kann also z. B. mit **`catch(Saeugetier&)`** alle Exceptions vom Typ Saeugetier und allen davon erbenden Klassen fangen.

universität wien

dynamic_cast

- Die umgekehrte Richtung der Typumwandlung kann explizit erzwungen werden, ist aber eventuell gefährlich:

```
(static_cast<Delphin *>(sp))->schwimmen();
(static_cast<Pferd &>(sr)).traben();
```

Syntaktisch korrekt, aber Objekt ist ein Delphin und kann nicht traben. Undefiniertes Vehalten zur Laufzeit.

- **`dynamic_cast`** führt eine Laufzeitüberprüfung durch, ob das Objekt auch wirklich den erforderlichen Typ hat und liefert im Fehlerfall entweder **`nullptr`** (für Pointer) oder wirft eine Exception vom Typ **`std::bad_cast`** (für Referenzen).

- `(dynamic_cast<Pferd &>(sr)).traben(); // Exception`

universität wien

Polymorphismus

- Die Auswirkung eines Funktionsaufrufs hängt von den Typen der beteiligten Objekte ab.

- Eine bereits bekannte Möglichkeit, dies zu erreichen, ist Überladen (ad-hoc Polymorphismus):

```
f(Saeugetier), f(Pferd), f(Delphin)
```

- Vererbung bietet eine weitere Möglichkeit:

```
class Saeugetier {
   …
   virtual void fortbewegen();
};

class Pferd: public Saeugetier {
   …
   void traben();
   virtual void fortbewegen() {traben();}
};

class Delphin: public Saeugetier {
  …
  void schwimmen();
  virtual void fortbewegen() {schwimmen();}
};
```

Muss nicht mehr explizit angegeben werden

```
Pferd *p{new Pferd};
Saeugetier *sp{p}; //erlaubt
sp->fortbewegen(); //trabt

Delphin *d{new Delphin};
Saeugetier &sr{*d};
sr.fortbewegen(); //schwimmt

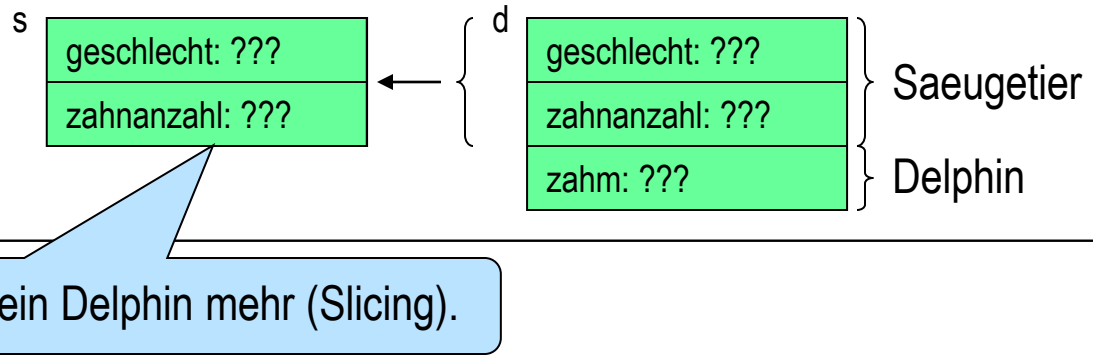
Saeugetier s{*p};
s.fortbewegen(); //Saeugetier::fortbewegen()
```

Polymorphes Verhalten der Objekte in C++ nur bei Zeigern und Referenzen.

Warum nur mit Pointern oder Referenzen? (1)

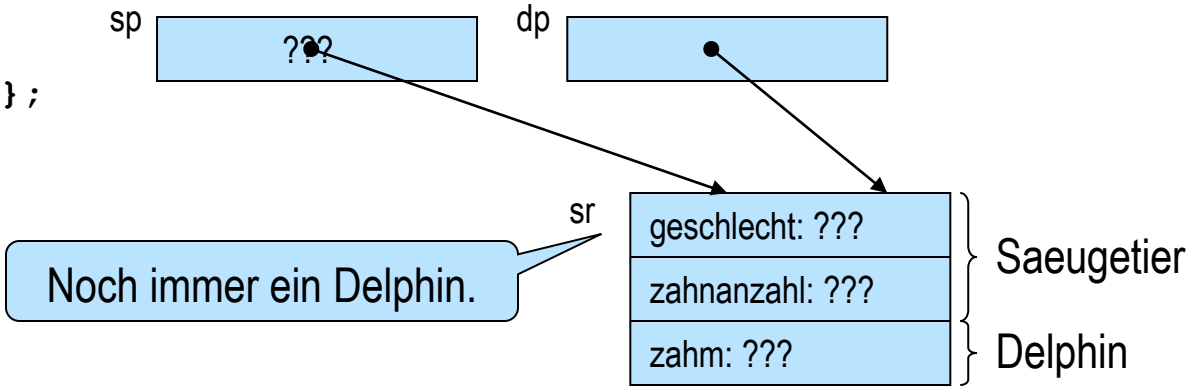
- Eine Variable in C++ entspricht einer Adresse, an der das Objekt im Speicher zu finden ist.
- Zuweisung entspricht einer Kopie von einer Adresse zu einer anderen.
- Objekte sind „modular“ aufgebaut (Eigene Instanzvariablen, Instanzvariablen der direkten Basisklasse und so weiter).
- Bei einer Zuweisung eines Objekts an eine Variable vom Typ einer Basisklasse, werden nur die relevanten Objektteile kopiert.

```
Saeugetier s;
Delphin d;
s = d;
```



Warum nur mit Pointern oder Referenzen? (2)

```
Saeugetier *sp;
Delphin *dp{new Delphin};
sp = dp;
Saeugetier &sr{*dp};
```



universität wien

override Spezifikation (Seit C++11)

• Damit ein Override stattfindet, müssen die Funktionssignaturen genau übereinstimmen. Um Irrtümer zu vermeiden kann explizit **`override`** angeführt werden:

```
class A {
   virtual int f() const;
};
class B : public A {
   int f(); //kein override (auch ein weiteres virtual hilft nicht)
   int f(double); //kein override
   int f() const; //override
}
class C : public A {
   int f() override;          //Fehler
   int f(double) override;    //Fehler
   int f() const override;    //OK
}
```

universität wien

Abstrakte Klasse

- Oft ist es nicht möglich, das Verhalten einer virtuellen Methode in der Basisklasse zu definieren. So haben z.B. Säugetiere keine gemeinsame Art sich fortzubewegen.
- Man kann die Methode dann als „pure virtual“ (abstrakt) kennzeichnen. Das heißt, sie wird erst in den abgeleiteten Klassen implementiert.
- Eine Klasse mit (mindestens einer) pure virtual Methode kann keine Objektinstanzen haben (z.B. es gibt kein Säugetier an sich, sondern es muss immer eine bestimmte Unterart sein). Eine solche Klasse wird als „abstrakt“ (abstract) bezeichnet.

Methode wird in dieser Klasse nicht implementiert. Erbende Klassen müssen (so sie nicht auch abstrakt sind) die Methode implementieren

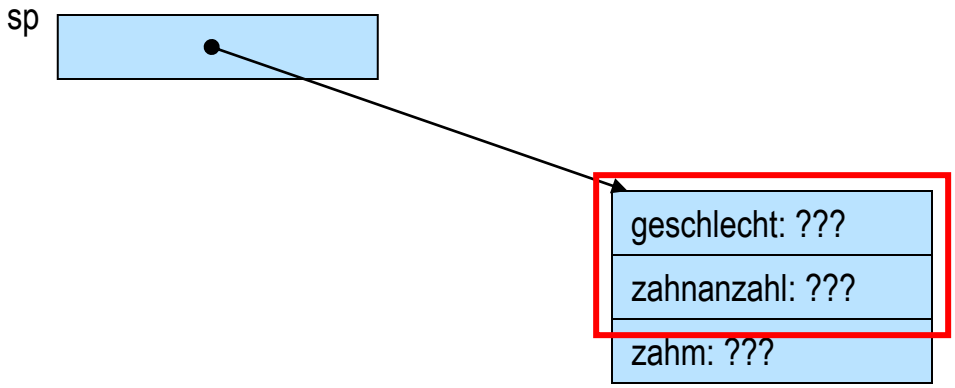
```
class Saeugetier {
   …
   virtual void fortbewegen() = 0;
};
```

universität wien

Virtueller Destruktor (1)

• Wird ein dynamisch erzeugtes Objekt über einen Pointer auf eine Basisklasse gelöscht, „weiß“ der Compiler die Größe des zu löschenden Objekts nicht.

```
Saeugetier *sp{new Delphin};
delete sp;
```



universität wien

Virtueller Destruktor (2)

- Lösung: virtueller Destruktor

```
class Saeugetier {
   …
   virtual ~Saeugetier();
};
```

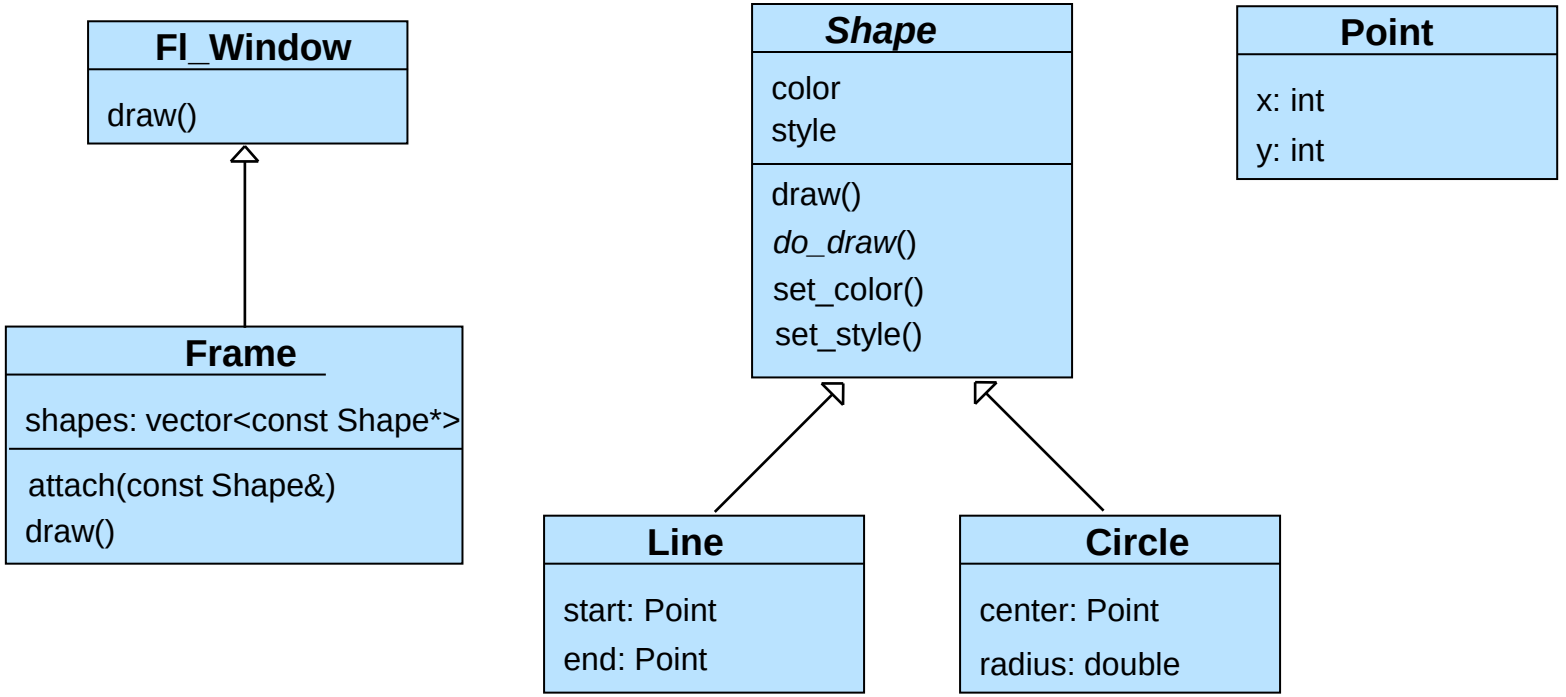
Damit haben alle erbenden Klassen ebenfalls einen virtuellen Destruktor (ob dort nochmals definiert oder nicht) und es wird bei polymorpher Verwendung immer der richtige Destruktor aufgerufen.

- Faustregel: In abstrakten Klassen immer einen virtuellen Destruktor definieren (meist schon in Klassen, sobald sie eine virtuelle Funktion haben).

universität wien

Beispiel: Einfache Grafikapplikation

- Kann unterschiedliche Formen (Shapes) zeichnen.



universität wien

Struct Point

```
struct Point {
  int x;
  int y;
  Point(int x, int y) : x(x), y(y) {};
};
```

universität wien

Klasse Frame

• Erbt von Fl_Window. Für dieses wird draw aufgerufen, wann immer der Inhalt des Fensters neu gezeichnet werden muss. Frame macht zuerst alles, was Fl_Window macht und ruft dann draw für jede assoziierte Form auf.

```
class Frame : public Fl_Window {
  std::vector<const Shape*> shapes;
public:
  Frame(int x, int y, const char* text) : Fl_Window(x, y, text) {}
  void draw() override {
    Fl_Window::draw();
    for (const auto& shape : shapes) shape->draw();
  }

  void attach(const Shape& s) {
    shapes.push_back(&s);
  }
};
```

universität wien

Klasse Shape

- Ruft, wenn draw() aufgerufen wird, die Methode do_draw() in der passenden erbenden Klasse auf. Erlaubt Setzen von Farbe und Linien-Stil.

```
class Shape {
  Fl_Color color {FL_BLACK};
  int style {0};
public:
  virtual ~Shape() {}
  void draw() const {
    Fl_Color old_color {fl_color()};
    fl_color(color);
    fl_line_style(style); //style cannot be queried - will be reset to default
    do_draw();
    fl_line_style(0); //default
    fl_color(old_color);
  }
  virtual void do_draw() const = 0;
  void set_color(Fl_Color color) {
     this->color = color;
  }
  void set_style(int style) {
     this->style = style;
  }
};
```

universität wien

Klassen Line und Circle

- "Zeichnen sich selbst".

```
class Line : public Shape {
  Point start;
  Point end;
public:
  Line(Point start, Point end) : start {start}, end {end} {}
  void do_draw() const override {
    fl_line(start.x, start.y, end.x, end.y);
  }
};

class Circle : public Shape {
  Point center;
  double radius;
public:
  Circle(Point center, double radius) : center {center}, radius {radius} {}
  void do_draw() const override {
    fl_circle(center.x, center.y, radius);
  }
};
```

universität wien

main()

```
int main() {
  Frame frame {450,220,"Graph"};
  Line line {Point{0, 0}, Point{100, 100}};
  frame.attach(line);
  Circle c1 {Point{0, 0}, 25};
  frame.attach(c1);
  Circle c2 {Point{100, 100}, 25};
  frame.attach(c2);
  c1.set_color(FL_RED);
  frame.show();
  return Fl::run();
}
```

universität wien

16 Lambda Expressions und Templates

universität wien

Sequenzen und Algorithmen

- Durch die Verwendung von Sequenzen lassen sich Algorithmen (wie z.B. das Finden des Minimums in einer Sequenz) unabhängig von den darunterliegenden Containertypen formulieren.

```
auto i = min_element(v.begin(),v.end()); //Minimum in v suchen
```

- Der Returnwert ist ein Iterator, der das gefundene Minimum referenziert, er ist gleich `v.end()`, wenn es kein Minimum gibt. Da der genaue Typ des Iterators vom zugrundeliegenden Container abhängt, wurde hier wieder `auto` verwendet.

universität wien

Funktionen als Parameter

- Bei der Formulierung von Algorithmen ist es durchaus nicht ungewöhnlich, dass Funktionen als Parameter verwendet werden.

```
auto i = find_if(v.begin(),v.end(),predicate);
```

- Soll einen Iterator liefern, der das erste Element in der Sequenz referenziert, für das der Aufruf der Funktion predicate true liefert.
- Funktionen als Parameter werden oft auch für sogenannte Callback-Funktionen (z.B. in Libraries, vor allem GUI) verwendet. Damit wird eine Funktion definiert, die aufgerufen werden soll, wenn ein bestimmtes Ereignis eintritt.)
- Es gibt mehrere Methoden, um Funktionen als Parameter zu definieren.

universität wien

1) Funktionspointer

- Schon in C verfügbar:

```
bool f(int val) {...}
bool (*predicate)(int) {f}; // auch &f möglich; Klammern essentiell
predicate(7); // Aufruf; auch (*predicate)(7) möglich;
auto i = find_if(v.begin(),v.end(),predicate);
```

- + mit allen Versionen von C, C++ verträglich
- - Syntax komplex
- - Funktion muss anderswo definiert werden (oft auch in einer anderen Datei).

universität wien

2) Funktionsobjekte (Funktoren)

• Objekte, in deren Klasse **`operator()`** überladen wurde. Sie können wie normale Objekte behandelt, aber auch wie Funktionen aufgerufen werden.

```
class Predicate {
  ...
public:
  bool operator()(int val) const {...};
};
Predicate predicate;
auto i = find_if(v.begin(),v.end(),predicate);
```

- + Objekt kann beliebig komplexen internen Zustand besitzen
- + Vorteile der objektorientierten Programmierung wie z.B. Kapselung, Vererbung
- + Syntax einfacher
- - Eine eigene Klasse muss definiert werden

universität wien

3) Lambdaausdrücke (lambda expressions)

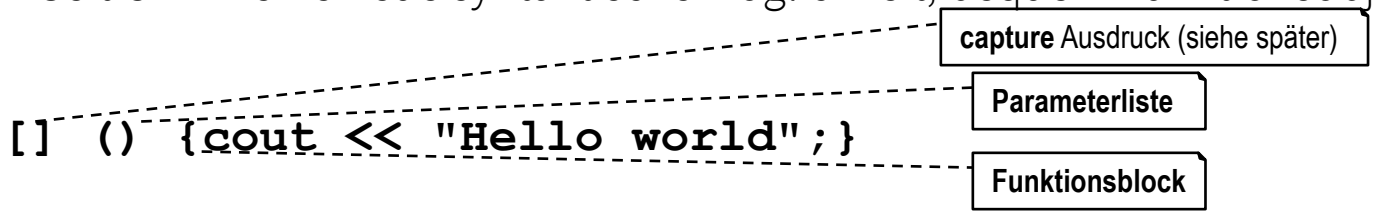
- Ein Ausdruck, der eine Funktion definiert.

```
auto i = find_if(v.begin(),v.end(),[] (int val) {...});
```

- + Definition an der Stelle der Verwendung
- + Syntax einfacher
- - Temporäres Objekt, daher kein dauerhafter Zustand

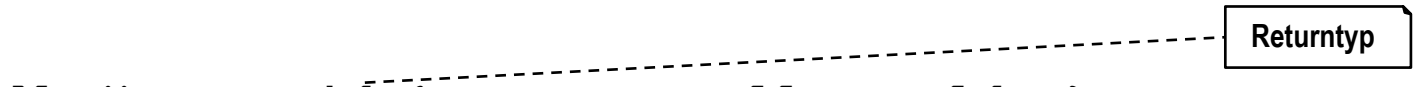
Lambdaexpressions Syntax (1)

- Seit C++11 eine neue syntaktische Möglichkeit, bequem Funktionsobjekte zu verwenden:




```
[] () {cout << "Hello world";}
```

- Der Typ des Returnwerts (im konkreten Fall void) kann vom Compiler automatisch ermittelt werden, soweit nicht mehrere Returnstatements im Block vorkommen (manche Compiler finden auch bei mehreren Returnstatements einen passenden Returntyp, aber der Standard garantiert das nicht).
- Angabe des Returntyps ist auch explizit möglich:




```
[] () -> void {cout << "Hello world";}
```

universität wien

Lambdaexpressions Syntax (2)

- Auch eine „exception specification“ kann angegeben werden. In C++11 gelten allerdings „exception specifications“ als überholt (deprecated) in C++17 wurden sie abgeschafft.

```
[] () throws() {cout << "Hello world";}
```

exception specification

- Auch noexcept ist möglich:

```
[] () noexcept {cout << "Hello world";}
```

Funktion darf keine Exception werfen

universität wien

Lambdaexpressions Closures

- Ein Lambda-Ausdruck kann Variablen, die in der umgebenden Funktion zugreifbar (in scope) sind, referenzieren (in einer Methode damit auch Instanzvariablen der entsprechenden Klasse)

```
class X {
  int x;
public:
  void methode() {
    int n;
    [=] {cout << n << x;}; //this->x
  }
};
```

Damit das funktioniert, muss das erzeugte Funktionsobjekt auch die gesamte (referenzierte) Umgebung beinhalten. Im Allgemeinen wird das erreicht, indem die entsprechenden Variablen vom Stack auf den Heap transferiert werden und dafür Sorge getragen wird, dass der Zugriff auf die Variablen von verschiedenen Seiten weiterhin funktioniert.
Das Funktionsobjekt schließt somit die umgebenden Variablen mit ein. Daher der Begriff **Closure**.

In C++ wird das ein wenig “hemdsärmelig” gelöst…

universität wien

Lambdaexpressions Capture Klausel

- Die „capture clause“ legt fest, wie die vom Lambda Ausdruck verwendeten Variablen im Funktionsobjekt gespeichert werden:

```
[]                  //kein capture – kein Zugriff auf Variablen der Umgebung
[var1, &var2, …]    //explizite Liste der zugreifbaren Variablen
                    //& bedeutet, dass Variable als Referenz übernommen wird
                    //nicht mehr verwendbar, wenn Variablen out of scope sind!
                    //andere Variable werden in konstante Instanzvariable des
                    //Funktionsobjekts kopiert (mutable notwendig, um diese
                    //ändern zu können)
[=, &var1, …]       //Defaultübernahme per Kopie, die explizit angegebenen
                    //Variablen müssen alle Referenzen sein
[&, var1, …] //Defaultübernahme als Referenz, die explizit angegebenen
                    //Variablen müssen alle Kopien sein
[this]              //Schlüsselwort this kann verwendet werden, um explizit
                    //Zugriff auf Instanzvariablen zu erhalten.
                    //this ist automatisch in = oder & inkludiert.
                    //Auch *this möglich (subtile Änderungen zwischen C++ Standards)
```

universität wien

Lambdaexpressions Typ

- Der Typ eines Lambda-Ausdrucks hat keine syntaktische Entsprechung. Um eine Variable, einen Parameter oder einen Returnwert vom passenden Typ zu definieren, wird daher auto, std::function oder ein Template verwendet:

```
#include<functional> //definiert std::function
std::function<int (int)> g(std::function<void ()> f) {
  f();
  return [] (int n) {return n*2;};
}
int main() {
  auto func{[] () -> void {cout << "Hello world\n";}};
  std::function<void ()> func2;
  func2 = func;
  auto h{g(func2)};
  cout << h(3) << '\n';
}
```

- **`std::function`** ist verträglich mit Lambdaausdrücken, Funktoren und Funktionspointern!

universität wien

Zurück zu find_if

- Wir hatten den Funktionsaufruf

```
auto i = find_if(v.begin(),v.end(),predicate);
```

- Aber wie soll so eine Funktion deklariert/definiert werden? Für **`std::vector<int>`** könnte

```
vector<int>::iterator find_if(vector<int>::iterator,
                              vector<int>::iterator,
                              std::function<bool(int)>);
```

oder

```
int* find_if(int*, int*, std::function<bool(int)>);
```

verwendet werden.

- Der Datentyp der Iteratoren ist aber vom verwendeten Container abhängig. Wir müssen also den Datentyp unserer Funktionsparameter parameterisieren.

universität wien

Templatefunktion find_if (1)

- Templates bilden die Möglichkeit, die Typen von Variablen und Funktionsparametern zu parametrisieren:

```
template<typename T> T find_if(T, T, std::function<bool(int)>);
```

- Statt typename kann (fast) synonym auch class verwendet werden (mit C++17 werden beide Schlüsselworte in diesem Kontext ganz synonym sein).

```
template<typename T> T find_if(T s, T e, std::function<bool(int)> p) {
  for (T i {s}; i != e; ++i) if (p(*i)) return i;
  return e;
}
```

- Der Compiler erkennt beim Aufruf der Funktion den Datentyp T und generiert den entsprechenden Code (dazu muss die Definition des Templatecodes in der inkludierten Header Datei zugreifbar sein).

universität wien

Templatefunktion find_if (2)

- Will man auch vom Typ der im jeweiligen Container gespeicherten Elemente unabhängig sein, so kann man einen weiteren Typparameter einführen:

```
template<typename T, typename P> T find_if(T, T, P);


template<typename T, typename P> T find_if(T s, T e, P p) {
  for (; s != e; ++s) if (p(*s)) return s; //Variable i kann eingespart werden
  return e;
}
```

- Die Datentypen T und P müssen selbstverständlich so beschaffen sein, dass die in der Funktion verwendeten Operationen mit ihnen ausführbar sind.

Templates ein simples Beispiel (1)

Templates bilden die Möglichkeit, die Typen von Variablen und Funktionsparametern zu parametrisieren (parametrischer Polymorphismus):

```
template <typename E>
E max(E a, E b) {
  return a>b ? a : b;
}
```

E ist ein Typparameter und wird bei Bedarf durch einen konkreten Typ ersetzt.
(Es können beliebig viele Typparameter mit Komma getrennt angegeben werden)

Dieses Template kann für alle Datentypen verwendet werden, die einen passenden Operator **>** unterstützen.

Verwendung:

```
max(7,5);
max(1.7,9.1);
max<int>(7,5);
max<long>(3,10.5);
max<double>(3,10.5);
…
max<const char*>("abc","xyz");
```

Problem!

universität wien

Templates ein simples Beispiel (2)

Für bestimmte Werte (Typen) von Typparametern können spezielle Versionen der Templatefunktion definiert werden:

Hier werden nur die Typparameter angeführt, die in dieser Spezialisierung nicht durch konkrete Typen ersetzt wurden

```
template <>
const char *max(const char *a, const char *b)
{
  return strcmp(a,b)>0 ? a : b;
}
```

Verwendung:

```
max("abc","xyz");
max<const char*>("abc","xyz");
```

universität wien

Template Klassen

- Wird eine Template Klasse verwendet, so sind alle Methoden der Klasse implizit Templatefunktionen

```
template<typename T>
class Vector {
  size_t max_sz;
  size_t sz;
  T* element;
public:
  void push_back(const T&);
  ...
};


template<typename T>
Vector<T>::push_back(const T& val) {...}
```

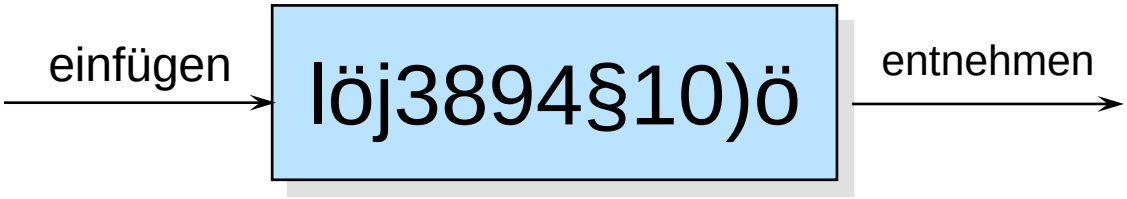
universität
wien

17 Stream I/O

universität wien

Streams

- Die Bezeichnung Stream entspringt der Vorstellung eines Stroms von Zeichen, in den man an einem Ende Zeichen einfügen und am anderen Ende Zeichen wieder entnehmen kann.



- Streams werden in der Regel mit Geräten (z.B.: Tastatur – **`cin`**, Bildschirm – **`cout`**) oder Dateien verbunden.

universität wien

Insertion und Extraktion

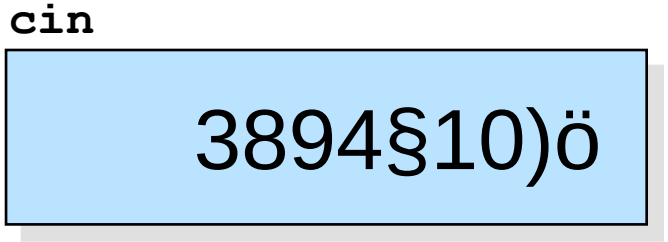
- In C++ kann das Einfügen in einen Stream mittels des Insertionsoperators **<<** durchgeführt werden.
- Die Entnahme aus einem Stream kann mittels Extraktionsoperator **>>** durchgeführt werden.
- Je nach Art des Streams können manchmal beide, oft aber auch nur jeweils eine der Operationen zulässig sein (IO-Stream, Inputstream z.B. **cin**, Outputstream z.B. **cout**)

```
char c;
cin >> c;  //Extraktion eines Zeichens
cout << c; //Insertion eines Zeichens
```

universität wien

Formatierte Ein/Ausgabe

- Werden andere Datentypen als Zeichen verwendet, so werden automatisch Konversionen durchgeführt.
- Beim Einlesen werden in der Regel Trennzeichen (white-space character) überlesen, so lange bis das erste Zeichen gefunden wird, das zum erwarteten Datentyp passt. Das Einlesen stoppt, wenn ein Zeichen nicht mehr zum erwarteten Datentyp passt (beim Datentyp Zeichenkette ist das wieder ein Trennzeichen).




```
int i;
cin>>i  //i erhält den Wert 3894
```

universität wien

Fehlerbehandlung (1)

- Tritt ein Fehler in einem Stream auf (z.B. es wird keine Ziffer beim Einlesen eines int-Wertes gefunden), so wird der Stream intern als ungültig markiert und alle weiteren Extraktions/Insertions-Operationen werden ignoriert.

```
int i;
do
  cin >> i;
while (i!=-1); //Endlosschleife, falls z.B.
               //das Zeichen x eingegeben
               //wird
```

universität wien

Fehlerbehandlung (2)

- Wird der Name eines Streams in einem Kontext verwendet, in dem ein boolescher Wert erwartet wird, wird implizit eine Typkonversion vorgenommen. Diese liefert **`false`**, falls der Stream als fehlerhaft markiert ist, **`true`** sonst.

```
int i;
do
  cin >> i;
while (cin && i!=-1); //Schleife endet, auch
                      //wenn das Zeichen x
                      //eingegeben wird
```

universität wien

Fehlerbehandlung (3)

- Um die Art des Fehlers zu erkennen, stehen die Methoden **`good`**, **`bad`**, **`fail`** und **`eof`** zur Verfügung:

```
cin.good() //kein Fehler aufgetreten.
cin.bad()  //fataler Fehler (z.B. physikalisch)
           //Stream unbrauchbar.
cin.fail() //Operation konnte nicht durchgeführt werden
           //(z.B. falscher Datentyp) Stream kann
           //weiter verwendet werden falls nicht auch
           //cin.bad true liefert.
cin.eof()  //EOF (Dateiende) wurde erreicht.
```

universität wien

Fehlerzustand

- Der aktuelle Zustand eines Streams wird in einer Bitmaske festgehalten, die durch alle In-/Outputoperationen aktualisiert wird.
- Folgende Bits sind definiert:

`std::ios_base::eofbit`	Dateiende wurde erreicht
`std::ios_base::failbit`	Fehler bei einer Operation
`std::ios_base::badbit`	Fataler Fehler
`std::ios_base::goodbit`	Keine Exception (Wert 0)

- Die Maske kann mit der Methode **`rdstate`** gelesen und mit den Methoden **`setstate`** und **`clear`** gesetzt werden. Durch Übergabe einer gewünschten Maske an **`clear`** kann diese auch gesetzt werden. **`setstate`** setzt die im Parameter gesetzten Bits zusätzlich zu anderen bereits vorher eventuell gesetzten.

universität wien

Fehler zurücksetzen

- Man kann den Stream mittels der Methode **`clear`** wieder in einen gültigen Zustand zurückversetzen:

```
int i;
char dummy;
do {
  cin>>i;
  if (cin.fail()) {
    cin.clear();
    cin>>dummy;
  }
  else cout<<i;
} while (i!=-1);
```

universität wien

Die Methoden eof und good

- Die Methode **`good`** liefert genau dann **`true`**, wenn kein Fehler passiert ist (**`fail`**, **`bad`** und **`eof`** liefern alle **`false`**), **`false`** sonst.
- Die Methode **`eof`** liefert **`true`**, wenn das Ende des Streams erreicht wurde (genauer, wenn versucht wurde über das Ende des Streams hinaus zu lesen), **`false`** sonst.

universität wien

Fehlerbehandlung mittels Exceptions

- Man kann für jeden Stream mit der Methode **`exceptions`** festlegen, dass im Falle des Auftretens bestimmter Zustände eine Exception geworfen werden soll.
- Die geworfene Exception hat den Datentyp **`std::ios_base::failure`**. Im entsprechenden catch-Block kann die Fehlerursache festgestellt und eventuell behoben werden.
- Ursachen, die eine Exception auslösen können, sind:

`std::ios_base::eofbit`
`std::ios_base::failbit`
`std::ios_base::badbit`
`std::ios_base::goodbit`

- Ein Aufruf von **`exceptions`** ohne Parameter liefert die aktuelle Maske, mit Parameter wird die Maske entsprechend gesetzt und der Fehlerzustand des Streams (wie durch einen Aufruf von **`clear`**) zurückgesetzt.

universität wien

Stream Exceptions Beispiel

bitweises logisches Oder

```
int i;
cin.exceptions(ios::failbit|ios::badbit);
try {
  do
    cin>>i;
  while (i!=-1);
}
catch (ios_base::failure){
  if (cin.fail()) cout<<"Bitte nur Zahlen eingeben"<<endl;
  else throw;
}
```

universität wien

Formatierte Ein-/Ausgabe

- Mit Hilfe sogenannter Manipulatoren lässt sich das Format der Ein- oder Ausgabe beeinflussen.
- z.B.: Zahlenbasis (für ganzzahlige Werte):

```
cout << std::dec << 1234; //1234
cout << std::hex << 1234; //4d2
cout << std::oct << 1234; //2322
cout << 1234; //2322 sticky!

cout << std::showbase; //Basen anzeigen

cout << 1234; //02322
cout << std::hex << 1234; //0x4d2
cout << std::dec << 1234; //1234
```

universität wien

Manipulatoren für Floating Point Werte

- Es wird eine (konfigurierbare) Anzahl n von Stellen ausgegeben, und zwar:

`defaultfloat`	wählt bestes Format mit n Stellen (default)
`scientific`	eine Ziffer vor dem Komma, n danach (mit Exponent)
`fixed`	n Ziffern nach dem Komma (kein Exponent)

- z.B.: Zahlenbasis (für ganzzahlige Werte):

```
cout << 1234.56789; //1234.57
cout << std::scientific;
cout << 1234.56789; //1.234568e+03
cout << std::fixed;
cout << 1234.56789; //1234.567890
```

Anzahl der Stellen n kann mit setprecision(n) festgelegt werden, default ist 6:

```
cout << std::setprecision(3);
cout << 1234.56789; //1234.568
```

universität wien

Manipulatoren für Feldbreite und Füllung

`setw` setzt die Breite, die für die nächste Ausgabe mindestens verwendet wird

`setfill` setzt das Zeichen, das zum Auffüllen verwendet wird

```
cout << std::setw(2) << "abcd"; //abcd
cout << std::setw(6) << "abcd"; //  abcd
cout << std::setfill('*');
cout << "abcd";                   //abcd
cout << std::setw(6) << "abcd"; //**abcd
cout << std::setw(6) << 12;     //****12
```

- Die meisten Manipulatoren sind in von **`iostream`** bereits inkludierten Header-Dateien deklariert. Für einige wenige (wie **`setw`** und **`setfill`**) muss **`iomanip`** inkludiert werden.

universität wien

Datei Ein/Ausgabe

- In der Library **`fstream`** werden zwei Datentypen für jeweils einen Output- und einen Input-Stream definiert, die direkt mit Dateien (Files) verbunden sind: **`ofstream`** und **`ifstream`**. Der Datentyp **`fstream`** dient für den Zugriff auf Dateien, die sowohl lesend, als auch schreibend verarbeitet werden sollen.

- Dateien werden geöffnet, indem man bei der Definition der Dateivariablen den gewünschten Dateinamen (eventuell inklusive Pfad) angibt:

```
ifstream inputFile{"MeineDatei.txt"};
```

- Dateien werden mittels der Methode close geschlossen. Diese wird auch automatisch im Destruktor aufgerufen.

```
inputFile.close();
```

universität wien

Datei open modes

- Durch Spezifikation eines open modes beim Öffnen einer Datei (zusätzlicher Parameter in Konstruktor bzw. **`open`**-Methode), kann man die genaue Arbeitsweise der Datei beeinflussen:

`std::ios_base::app`	append (am Ende anhängen)
`std::ios_base::ate`	at end (ans Ende positionieren)
`std::ios_base::binary`	Binärdatei (vs. Textdatei)
`std::ios_base::in`	input (lesend; default für ifstream)
`std::ios_base::out`	output (schreibend; default für ofstream)
`std::ios_base::trunc`	truncate (Inhalt der Datei verwerfen)

- z.B.:

```
ifstream inputFile("file.data",std::ios_base::out|std::ios_base::binary);
```

universität wien

Gültige Kombinationen von open modes

- Nicht alle Kombinationen von open modes sind sinnvoll oder erlaubt. Die erlaubten Kombinationen sind:

modestring	*openmode & ~ate*	*Action if file already exists*	*Action if file does not exist*
"r"	in	Read from start	Failure to open
"w"	out, out|trunc	Destroy contents	Create new
"a"	app, out|app	Append to file	Create new
"r+"	out|in	Read from start	Error
"w+"	out|in|trunc	Destroy contents	Create new
"a+"	out|in|app, in|app	Write to end	Create new
"rb"	binary|in	Read from start	Failure to open
"wb"	binary|out, binary|out|trunc	Destroy contents	Create new
"ab"	binary|app, binary|out|app	Write to end	Create new
"r+b"	binary|out|in	Read from start	Error
"w+b"	binary|out|in|trunc	Destroy contents	Create new
"a+b"	binary|out|in|app, binary|in|app	Write to end	Create new

Quelle: http://en.cppreference.com/w/cpp/io/basic_filebuf/open

universität wien

Beispiel zu Datei Ein/Ausgabe

```
#include<fstream>
ofstream outputFile{"MeineDatei.txt"};
outputFile<<1<<2<<3<<'x';
outputFile.close();
ifstream inputFile{"MeineDatei.txt"};
int in;
char c;
inputFile>>in>>c;
cout<<in<<c<<endl;
```

universität wien

Binary I/O

- Für binäre I/O-Operationen werden meist write- und read-Methoden genutzt, sowie oft auch Operationen um sich im stream zu positionieren:

```
read(char *buf, int count);  //lese count bytes nach buf
write(char *buf, int count); //schreibe count bytes von buf

seekg(int pos); //positioniere Leseposition(get) auf pos
seekp(int pos); //positioniere Schreibeposition (put) auf pos
tellg();        //retourniert aktuelle Leseposition
tellp();        //retourniert aktuelle Schreibeposition
                //Bytes werden von 0 beginnend gezählt.
```

- Gibt man bei den seek-Methoden einen weiteren Parameter an (**`std::ios_base::beg`**, **`std::ios_base::cur`** oder **`std::ios_base::end`**), so wird **`pos`** relativ zur entsprechenden Position interpretiert.

universität wien

Gepufferte Ein/Ausgabe (buffered I/O)

- Aus Effizienzgründen arbeiten Streams in der Regel gepuffert (**`cerr`** ist aber z.B. eine Ausnahme). Das heißt, die Änderungen werden nicht sofort an die assoziierte Hardware weitergegeben, sondern in einem Zwischenpuffer im Speicher des Rechners durchgeführt.
- Die Änderungen werden weitergegeben, wenn der Puffer voll ist, der Stream geschlossen wird oder einer der Manipulatoren **`flush`** oder **`endl`** verwendet wird.

universität wien

Stream Tying

```
std::basic_ostream<CharT,Traits>* tie();
std::basic_ostream<CharT,Traits>* tie( std::basic_ostream<CharT,Traits>* str );
```

- Um eine störungsfreie Zusammenarbeit von Streams zu ermöglichen, ist es möglich, an einen Stream mittels Aufruf der Methode **`tie`** einen Output-Stream zu binden. Vor jeder Ein-/Ausgabeoperation auf dem Stream wird dann zunächst **`flush`** für den angebundenen Stream aufgerufen. **`cout`** ist in dieser Weise an **`cin`** gebunden. So erscheinen Inputprompts auf dem Bildschirm sicher bevor die Eingabeoperation beginnt, z.B.:

```
cout << "Geben Sie bitte eine Zahl ein: ";
cin >> number;
```

- **`tie`** liefert als Returnwert einen Pointer auf den zuvor angebundenen Stream bzw. **`nullptr`**.

universität wien

String Streams

- Die Klassen **`stringstream`**, **`istringstream`** und **`ostringstream`** repräsentieren Streams, die nicht auf Geräten oder Dateien operieren, sondern auf Strings, die im Programm als Variablen angelegt werden. Damit sind zum Beispiel Konversionsoperationen möglich, z.B.:

```
#include<sstream>
std::string in{"17.2"};
std::string out{"Ergebnis:"};
int no;
char c;
std::istringstream read{in};
std::ostringstream write{out}; //verwendet eine Kopie von out
while (read >> no) {
  write << no*2;
  read >> c;
}
cout << out << '\n' << write.str() << '\n'; //Ausgabe: Ergebnis:\n344ebnis:
```

universität
wien

18 Eine eigene Vektor Klasse

universität wien

Phase 1:

- Ziel: Implementierung einer Klasse Vector analog zu std::vector<double>

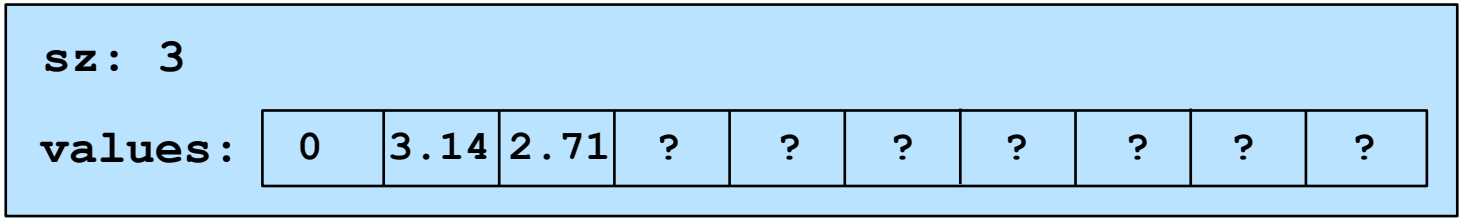
- In einer ersten Version beschränken wir uns auf Vektoren mit einer fixen maximalen Größe. Diese wird in der Header-Datei als globale Konstante **`vector_max_size`** vom Type **`size_t`** definiert.

- Für einen Vektor werden zwei Instanzvariablen benötigt:
- Ein double-Array **`values`** der Größe **`vector_max_size`**, um die Werte zu speichern.
- Ein Wert **`sz`** vom Typ **`size_t`**, der die aktuell gespeicherte Anzahl von Werten enthält.

universität wien

Ein Vektor (schematisch)

`vector_max_size: 10`



Zu allen Zeitpunkten müssen folgende Integritätsbedingungen gelten:

- $\mathbf{0} \leq \boldsymbol{sz} \leq \boldsymbol{vector\_max\_size}$
- Die gespeicherten double Werte sind $\boldsymbol{values}[\mathbf{0}] \ \ldots \boldsymbol{values}[\boldsymbol{sz} - \mathbf{1}]$
- $\boldsymbol{sz}$ gibt den nächsten freien Index im Feld $\boldsymbol{values}$ an.

universität wien

Funktionalität

- Methoden:
- Konstruktor: erstellt einen leeren Vektor
- **`size_t size() const:`** liefert die Anzahl der gespeicherten Elemente
- **`bool empty() const:`** liefert **`true`**, falls der Vektor leer ist, **`false`** sonst
- **`void push_back(double):`** fügt einen Wert am Ende ein; Exception, falls der Vektor schon voll ist.
- **`void pop_back():`** entfernt den Wert am Ende; Exception, falls der Vektor schon leer ist.
- **`double& operator[](size_t):`** liefert eine Referenz auf den indizierten Wert, bzw. eine Exception, falls der Index ungültig ist.
- **`void clear():`** entfernt alle Werte aus dem Vektor.
- **`ostream& print(ostream &) const:`** Gibt die Werte in Form einer Komma-separierten Liste (z.B.: [1, 2, 3]) aus.

- globale Funktionen:
- **`bool operator==(const Vector& lop, const Vector& rop): true`**, falls der Inhalt der beiden Vektoren exakt übereinstimmt
- **`bool operator>(const Vector& lop, const Vector& rop): true`**, falls rop in der lexikographischen Anordnung vor lop kommt.
- Analog für **`operator!=`**, **`operator>=`**, **`operator<`** und **`operator<=`**.

universität wien

Vorgehensweise

1. Erstellen Sie die Klassendefinition mit dem erforderlichen Interface in einer Datei vector.h.
2. Erstellen Sie die Implementierung der Klasse in einer Datei vector.cpp.
3. Erstellen Sie ein Hauptprogramm, in dem Sie die Funktionalität der Klasse testen.

universität wien

Problem

- Versuchen Sie folgende Funktion zu Ihrem Testprogramm hinzuzufügen:

```
void foo(const Vector& v) {
  cout<<v[0];
}
```

1. Welchen Fehler erhalten Sie?
2. Was ist die Ursache?
3. Wie kann man diesen Fehler beheben?
4. Passen Sie Ihre Klassendefinition (vector.h) und –implementierung (vector.cpp) entsprechend an.

universität wien

Erweiterungen

- Implementieren Sie Funktionen, um die Summe, das Minimum, das Maximum und den Mittelwert der gespeicherten Werte zu ermitteln (Warum gibt es derartige Funktionen nicht in **`std::vector`**?).

- Überladen Sie **`operator<<,`** sodass die in einem Vektor gespeicherten Werte in einer Komma-separierten Liste ausgegeben werden, z.B.: [1, 2, 0.5]

- Implementieren Sie Methoden **`insert(size_t, double)`** und **`erase(size_t),`** die einen Wert an einer vorgegebenen Stelle einfügen, bzw. von einer vorgegebenen Stelle entfernen.

- Anmerkung: Für diese Erweiterungen sind die Signaturen der Funktionen nicht mehr genau vorgegeben. Überlegen Sie sich selbst sinnvolle Returnwerte und Prototypen.

universität wien

Spezialaufgabe

• Implementieren Sie einen Konstruktor, der es erlaubt, Vektoren mit einer Liste von Werten zu initialisieren, z.B.:

```
Vector v {1, 2, 3};
```

• Dazu benötigen Sie einen Konstruktor, der eine initializer_list als Parameter akzeptiert. Das wurde in der Vorlesung nicht behandelt. Fragen Sie in der Übungsgruppe oder im Forum nach, bzw. googeln Sie, wie ein solcher Konstruktur aussieht.

universität wien

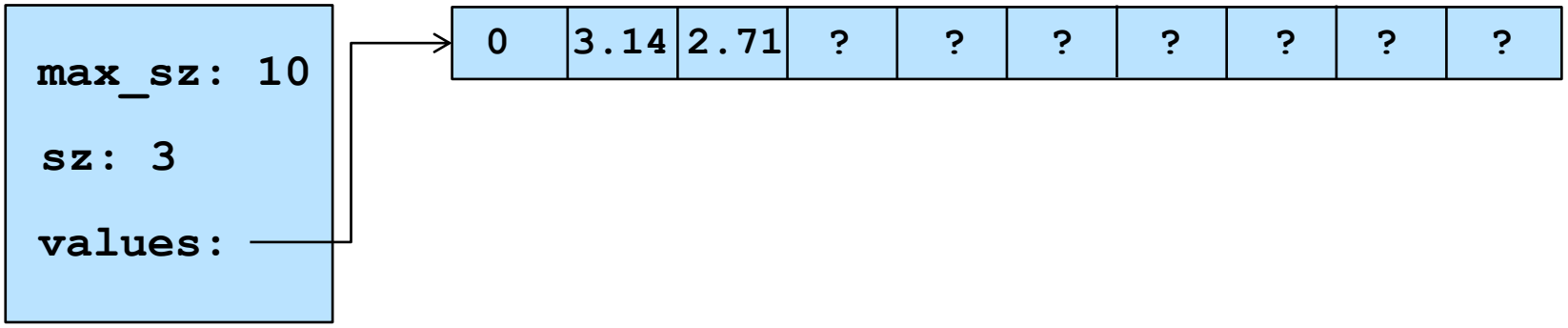
Phase 2

- Ziel: Die Klasse Vector soll bei Bedarf dynamisch wachsen.

- Wachstumsstrategie: Wenn für ein neu einzufügendes Element kein Platz mehr ist, so wird ein neuer Speicherblock mit doppelter Größe alloziert, die Inhalte des alten Blocks werden in den neuen Block kopiert und der alte Block wird freigegeben. Das einzufügende Element kann nun am nächsten verfügbaren Platz eingetragen werden.

- Statt der konstanten maximalen Größe **`vector_max_size`** wird nun eine Instanzvariable **`max_sz`** verwendet, die die aktuelle Größe des allozierten Speicherbereichs angibt.
- Statt eines statischen Arrays **`values`** wird ein Pointer **`values`** auf ein dynamisch alloziertes Array der Größe **`max_sz`** verwendet.
- (Größe ist hier immer in Einheiten von double-Werten zu verstehen.)

Ein dynamischer Vektor (schematisch)

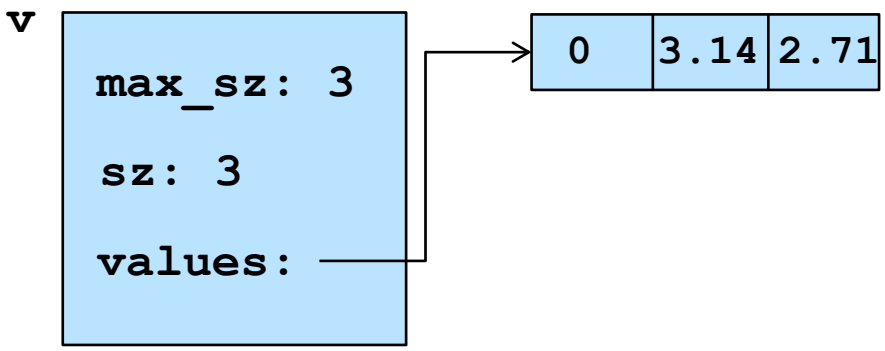


Zu allen Zeitpunkten müssen folgende Integritätsbedingungen gelten:

- $0 \leq sz \leq max\_sz$
- $values$ ist ein dynamisch allozierter Speicherberich der Größe max_sz
- Die gespeicherten double Werte sind $values[0] \ldots values[sz-1]$
- sz gibt den nächsten freien Index im Feld $values$ an.

universität wien

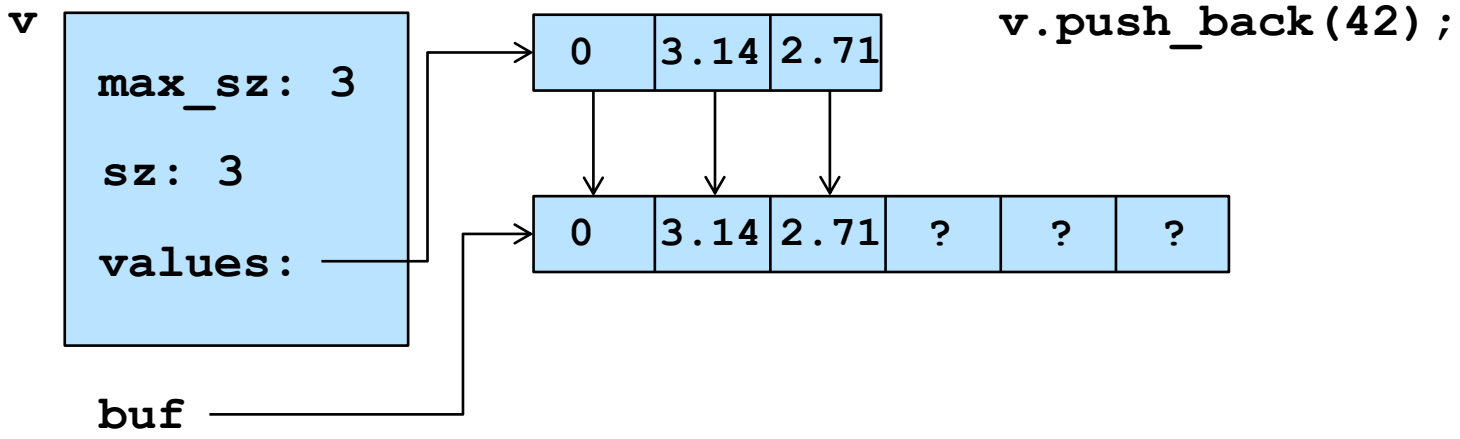
Einfügen und Vergrößern (1)




```
v.push_back(42);
```

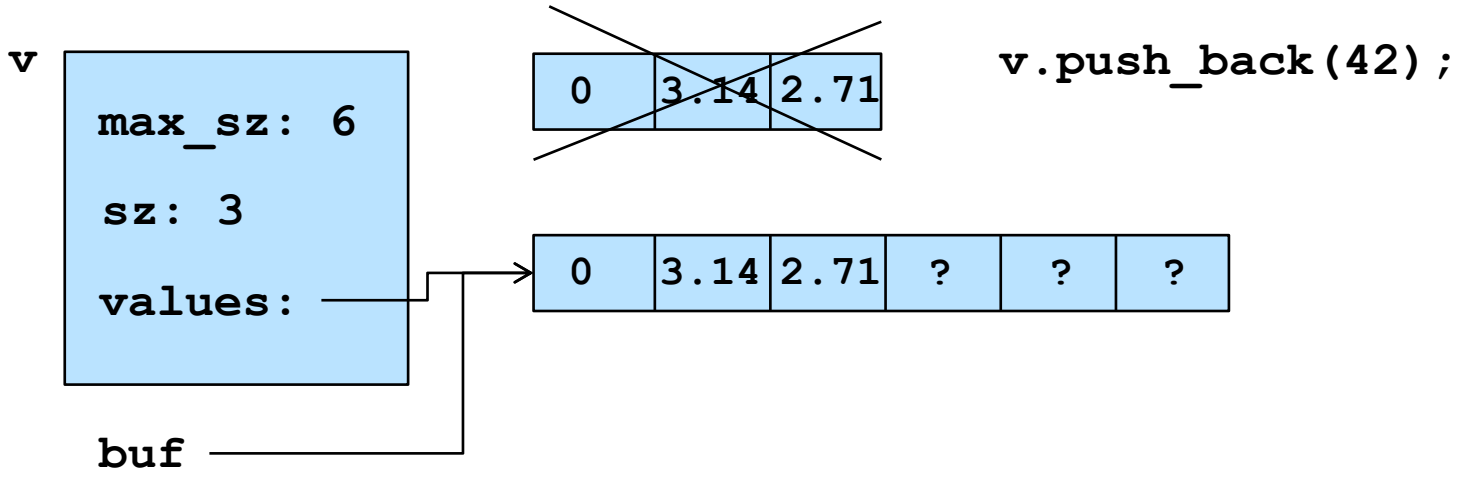
universität wien

Einfügen und Vergrößern (2)



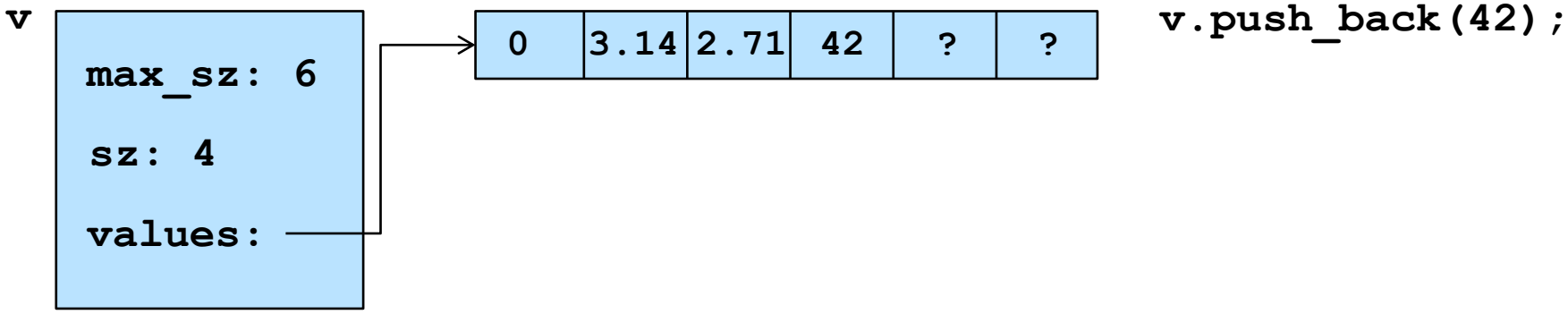
universität wien

Einfügen und Vergrößern (3)



universität wien

Einfügen und Vergrößern (4)



universität wien

Änderungen in der Klasse Vector

1) Instanzvariablen ändern / hinzufügen.
2) Neue Methoden implementieren / vorhandene Methoden anpassen (Da Arrays und Pointer verträglich sind, müssen nicht alle schon vorhandenen Methoden angepasst werden).
3) Zusätzliche Funktionalität implementieren.

universität wien

Methoden `reserve` und `shrink_to_fit`

- Da das Vergrößern des Speicherbereichs an unterschiedlichen Stellen benötigt wird (**`push_back`**, **`insert`**), empfiehlt es sich, eine Methode dafür zu schreiben:
- **`void reserve(size_t)`**: Vergrößert den Puffer, sodass mindestens so viele Elemente gespeichert werden können, wie der Parameter angibt (Eine Verkleinerung findet nicht statt).

- Um überflüssigen Puffer wieder freigeben zu können, wird eine Methode **`void shrink_to_fit()`** implementiert, die den Puffer so weit reduziert, dass gerade noch die aktuell im Vektor gespeicherten Elemente Platz haben (falls eine minimale Anzahl von Elementen für die Klasse definiert wird, dann darf diese hier selbstverständlich nicht unterschritten werden).

universität wien

Methoden, die geändert werden müssen

- Konstruktoren: müssen Speicher für die Elemente reservieren. Entweder reserviert man immer zumindest eine minimale Anzahl an Elementen (diese kann in einer Klassenvariable vorgegeben werden) oder man lässt auch 0 für die Puffergröße zu (Das muss dann beim Vergrößern entsprechend berücksichtigt werden, da Verdoppeln von 0 nicht zu einer Vergrößerung führt).
- **`Vector(size_t)`**: Ein zusätzlicher Konstruktor, der es erlaubt, die zu reservierende Anzahl von Elementen im Parameter anzugeben.
- `void push_back(double)`: fügt einen Wert am Ende ein; Vergrößern, falls der Vektor schon voll ist (keine Exception).
- `void insert(size_t, double)`: fügt einen Wert an der vorgegebenen Position ein; Vergrößern, falls der Vektor schon voll ist.

universität wien

Destruktor, Kopierkonstruktor, Kopierzuweisungsoperator

- Es ist der Destruktor für die Klasse Vector zu implementieren. Dieser muss reservierten Speicher wieder freigeben.
- Für ein ordnungsgemäßes Funktionieren sind auch die Implementierung eines Kopierkonstruktors und eines Kopierzuweisungsoperators notwendig.

universität wien

Weitere Funktionalität (globale Funktionen)

- Überladen Sie **`operator>>`**, sodass Vektoren eingelesen werden können. Das Eingabeformat soll dabei dem Ausgabeformat (also z.B [1,2,3]) entsprechen. Was passiert, wenn der leere Vektor ([]) eingegeben wird? Können Sie das beheben?

- Überladen Sie **`operator+`**, sodass zwei Vektoren miteinander mit + verknüpft werden können. Das Ergebnis der Operation soll ein Vektor sein, der alle Werte der beiden Vektor beinhaltet ("Zusammenhängen der beiden Vektoren" z.B.: [1,2,3]+[1,5,4] = [1,2,3,1,5,4]). Die beiden Operanden sollen nicht verändert werden.

- Erstellen Sie eine Funktion **`Vector find(Vector srch, Vector pttrn)`**, die einen Vektor mit allen Indizes liefert, an denen der Vektor pttrn im Vektor srch auftritt.
- z.B.: find([1,2,3,4,2,3,1,2,3], [2,3]) = [1,4,7], find([1,2,3],[7])=[]
- Was sollte eine Suche nach einem leeren Vektor (find([1,2,3],[])) liefern?

universität wien

Verwendung

• Verwenden Sie Ihre Klasse Vector, um eine Klasse zu implementieren, die die aus der Mathematik bekannten Vektoren im $\mathbb{R}^n$ repräsentieren. Überladen Sie Operatoren um die üblichen Operationen (Addition, Subtraktion, Multiplikation mit einem Skalar, Skalarprodukt und Vektorprodukt) mit solchen Vektoren ausführen zu können.

universität wien

Phase 3

- Ziel: Speichern beliebiger Datentypen in einem Objekt vom Typ Vector.

- Umschreiben der Vector-Klasse in ein Template, das den gespeicherten Datentyp als Template-Parameter hat.

- Testen der erstellten Klasse. Vergleich mit std::vector durchführen.

universität wien

Anhang

universität wien

Anhang: Escape-Sequenzen

`\n`	Neue Zeile (new line)
`\r`	Wagenrücklauf (carriage return)
`\t`	Horizontaler Tabulator
`\v`	Vertikaler Tabulator
`\b`	Backspace
`\f`	Neue Seite (formfeed)
`\a`	Alarm
`\\`	Backslash
`\'`	Apostroph
`\"`	Anführungszeichen
`\041`	Zeichencode im Oktalsystem (Basis 8)
`\x4F`	Zeichencode im Hexadezimalsystem (Basis 16)

universität wien

Anhang: Schlüsselwörter in C++

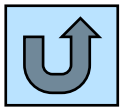
alignas (since C++11)
alignof (since C++11)
and
and_eq
asm
atomic_cancel (TM TS)
atomic_commit (TM TS)
atomic_noexcept (TM TS)
auto(1)
bitand
bitor
bool
break
case
catch
char
char8_t (since C++20)
char16_t (since C++11)
char32_t (since C++11)
class(1)
compl
concept (since C++20)
const
consteval (since C++20)
constexpr (since C++11)
constinit (since C++20)
const_cast
continue
co_await (since C++20)
co_return (since C++20)
co_yield (since C++20)
decltype (since C++11)
default(1)
delete(1)
do
double
dynamic_cast
else
enum
explicit
export(1)(3)
extern(1)
false
float
for
friend
goto
if
inline(1)
int
long
mutable(1)
namespace
new
noexcept (since C++11)
not
not_eq
nullptr (since C++11)
operator
or
or_eq
private
protected
public
reflexpr (reflection TS)
register(2)
reinterpret_cast
requires (since C++20)
return
short
signed
sizeof(1)
static
static_assert (since C++11)
static_cast
struct(1)
switch
synchronized (TM TS)
template
this
thread_local (since C++11)
throw
true
try
typedef
typeid
typename
union
unsigned
using(1)
virtual
void
volatile
wchar_t
while
xor
xor_eq

Identifiers with special meaning:
Keine Schlüsselwörter, haben aber in bestimmtem Kontext spezielle Bedeutung

override (C++11)
final (C++11)
import (C++20)
module (C++20)
transaction_safe (TM TS)
transaction_safe_dynamic (TM TS)

(1) **neue oder geänderte Bedeutung mit C++11**
(2) **geänderte Bedeutung mit C++17**
(3) **geänderte Bedeutung mit C++20**

Quelle: http://en.cppreference.com/w/cpp/keyword



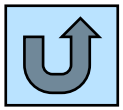
universität wien

Anhang: Die dunkle Seite von C++

- Undefiniertes Verhalten (undefined behavior):
  - Beispiel: nicht initialisierte Variable
  - Es kann *alles* passieren
  - Unbedingt vermeiden!

- Unspezifiziertes Verhalten (unspecified):
  - Beispiel Auswertungsreihenfolge bei Ausdrücken (sin(x)+cos(y))
  - Überraschungen möglich
  - Programm so formulieren, dass es nicht von unspezifiziertem Verhalten abhängt.

- Implementationsabhängiges Verhalten (implementation defined):
  - Beispiel: Größe von Datentypen
  - Nicht vom Standard definiert
  - Die für den jeweiligen Compiler gewählten Werte müssen dokumentiert werden
  - Programme möglichst portabel formulieren



universität wien

Anhang: Suffixe für Literale

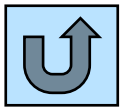
Types allowed for integer literals

suffix	decimal bases	hexadecimal or octal bases
no suffix	int long int long long int(since C++11)	Int unsigned int long int unsigned long int long long int(since C++11) unsigned long long int(since C++11)
u or U	unsigned int unsigned long int unsigned long long int(since C++11)	unsigned int unsigned long int unsigned long long int(since C++11)
l or L	long int unsigned long int(until C++11) long long int(since C++11)	long int unsigned long int long long int(since C++11) unsigned long long int(since C++11)
both l/L and u/U	unsigned long int unsigned long long int(since C++11)	unsigned long int(since C++11) unsigned long long int(since C++11)
ll or LL	long long int(since C++11)	long long int(since C++11) unsigned long long int(since C++11)
both ll/LL and u/U	unsigned long long int(since C++11)	unsigned long long int(since C++11)

Types allowed for floating point literals

(no suffix) defines double
f F defines float
l L defines long double

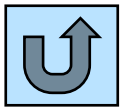
Quelle: http://en.cppreference.com/w/cpp/keyword



universität wien

Anhang: Namen (identifiers)

- Namen in C++ sind case-sensitive.
  - Main() ist nicht dasselbe wie main()
- Namen müssen mit einem Buchstaben oder _ beginnen und nur Buchstaben, Ziffern und _ beinhalten.
  - Namen die mit _ beginnen, bzw. __ enthalten, werden von C++ intern verwendet und sollten vermieden werden.
  - (Der aktuelle C++ Standard lässt außer Buchstaben auch viele Unicode-Zeichen zu, nicht alle Compiler unterstützen dies jedoch vollständig.)
- Schlüsselwörter sind nicht als Namen erlaubt.
- Konventionen:
  - Möglichst “sprechende” Namen verwenden
  - Namen aus mehreren Wörtern mit _ trennen (z.B. my_var)
  - Variablennamen mit Kleinbuchstaben beginnen



universität wien

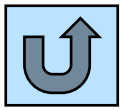
Anhang: Initialisierungssyntax

- Wir verwenden hier die modernere uniform initialization (auch universal initialization), die mit C++11 definiert wurde.

- Die ältere Version (int x = 3;) sollte man ebenfalls lesen können. Sie wird z.B. auch von Stroustrup verwendet.

- Uniform initialization verbietet implizite Typumwandlungen, bei denen Informationen verloren gehen könnten (narrowing conversion)

```
int x = 3.5; //OK eventuell Warnung des Compilers
int y {3.5}; //Fehler
int z {static_cast<int>(3.5)}; //OK explizite Typumwandlung
```



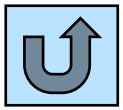
universität wien

Anhang: Seiteneffekt (side effect)

- Änderung des Programmzustands (der Werte von Variablen) bei der Ausführung von Programmstatements.

- In C++ bei der Zuweisung und bei Operatoren, die den Wert von Variablen ändern sollen (z.B. ++) erwünscht.

- Sonst im Allgemeinen unerwünscht.



universität wien

Anhang: Operatoren Übersicht (1)

- Die folgende Tabelle enthält alle C++ Operatoren. Die Bindungsstärke (precedence) aller Operatoren in einem Kasten ist gleich. Je weiter unten der Kasten, desto kleiner die Bindungsstärke.
- In der letzten Spalte ist vermerkt, ob die Operatoren der jeweiligen Gruppe rechts- oder linksassoziativ sind.

operator	name	example	
:: ::	scope resolution global	Class::member ::variable	right

universität wien

Anhang: Operatoren Übersicht (2)

operator	name	example	
.	member selection	object.member	left
->	member selection	object->member	
[ ]	subscripting	array[index]	
( )	function call	function (parameters)	
( )	value construction	type(expressions)	
++	post increment	lvalue++	
--	post decrement	lvalue--	
typeid()	run time type information	typeid(val)	
type() type{}	functional cast	int(7.3)	
const_cast	type cast	const_cast<int>x	
dynamic_cast	type cast	dynamic_cast<int>x	
reinterpret_cast	type cast	reinterpret_cast<char *>x	
static_cast	type cast	static_cast<int>x	

universität wien

Anhang: Operatoren Übersicht (3)

operator	name	example	
sizeof	size of object	sizeof expr	
sizeof()	size of type	sizeof (type)	
++	pre increment	++lvalue	
--	pre decrement	--lvalue	
~	complement	~expr	
!	not	!expr	
-	unary -	-expr	
+	unary +	+expr	
&	address of	&lvalue	right
*	dereference	*expr	
new	create (allocate)	new type	
delete	destroy (de-allocate)	delete pointer	
delete[ ]	destroy array	delete[ ] pointer	
co_await	await-expression (C++20)		
( )	cast	(type) expr	

universität wien

Anhang: Operatoren Übersicht (4)

operator	name	example	
.* ->*	member selection member selection	object.*pointer_to_member pointer->*pointer_to_member	left
* / %	multiply divide modulo	expr*expr expr/expr expr%expr	left
+ -	add subtract	expr+expr expr-expr	left
<< >>	shift left shift right	expr<<expr expr>>expr	left
<=>	three-way comparison (since C++20)	expr<=>expr	left

universität wien

Anhang: Operatoren Übersicht (5)

operator	name	example	
<=>	threeway comparison	expr<=>expr (spaceship operator ab C++20)	left
< <= > >=	less than less or equal greater than greater or equal	expr<expr expr<=expr expr>expr expr>=expr	left
== !=	equal not equal	expr==expr expr!=expr	left
&	bitwise AND	expr&expr	left
^	bitwise exclusive OR	expr^expr	left
|	bitwise inclusive OR	expr|expr	left

universität wien

Anhang: Operatoren Übersicht (6)

operator	name	example	
&&	logical AND	expr&&expr	left
||	logical OR	expr||expr	left

universität wien

Anhang: Operatoren Übersicht (7)

operator	name	example	
?:	ternary conditional	expr?expr:expr	
throw	throw exception	throw runtime_error("msg")	
co_yield	yield expression (C++20)		
=	simple assignment	lvalue=expr	
*=	multiply and assign	lvalue*=expr	
/=	divide and assign	lvalue/=expr	
%=	modulo and assign	lvalue%=expr	right
+=	add and assign	lvalue+=expr	
-=	subtract and assign	lvalue-=expr	
<<=	shift left and assign	lvalue<<=expr	
>>=	shift right and assign	lvalue>>=expr	
&=	AND and assign	lvalue&=expr	
|=	OR and assign	lvalue|=expr	
^=	XOR and assign	lvalue^=expr	

universität wien

Anhang: Operatoren Übersicht (8)

operator	name	example	
,	sequencing	expr,expr	left

universität wien

Anhang: Extended Backus-Naur Form (EBNF)

- Sprache zur Formulierung von Syntaxregeln.

- Terminalsymbole (Orthosymbole) `'if'`, `'int'`, `'('`, `'{'`... sind Worte, die in der Sprache auftreten.

- Metasymbole werden durch Produktionsregeln aufgelöst:

  assignment_operator = `'='` | `'+='` | `'-='`

|	Auswahl
[ ]	optional
{ }	beliebig oft (auch 0 mal)
( )	Gruppierung

universität wien

Anhang: Vektor mit Index traversieren (1)

- Jeder Vektor-Typ hat im Prinzip einen eigenen Datentyp für seine Indizes. Dieser ist groß genug, um alle Elemente eines beliebig großen entsprechenden Vektors zu halten:

```
vector<T>::size_type
```

- Die Schleife sollte also eigentlich so aussehen:

```
int sum {0};
for (vector<int>::size_type i {0}; i < v.size(); ++i)
  sum += v.at(i);
cout << sum << '\n';
```

- Für Vektoren aus der Standardlibrary ist aber **`size_t`** ausreichend. (**`size_t`** kann die Größe beliebiger zusammenhängender Objekte im Speicher enthalten.)

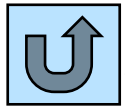
universität wien

Anhang: Vektor mit Index traversieren

- In jedem Schleifendurchlauf die Methode size aufzurufen, scheint ineffizient. Man könnte optimieren:

```
int sum {0};
for (size_t i {0}, stop {v.size()}; i < stop; ++i)
  sum += v.at(i);
cout << sum << '\n';
```

- Für die Vektoren der Standardlibrary kann der Compiler aber den Aufruf von **`size`** gut optimieren. Wir verwenden daher die besser lesbare Form. (Die „handoptimierte Version schlägt außerdem fehl, wenn sich die Länge des Vektors während der Schleife verändert.)

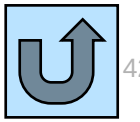


universität wien

Anhang: Trennzeichen (white-space characters)

In der Regel werden die folgenden Zeichen als Trennzeichen betrachtet. (Das kann aber durch länderspezifische Einstellungen sogenannte locale settings modifiziert werden.)

' '	(0x20)	space (SPC)
'\t'	(0x09)	horizontal tab (TAB)
'\n'	(0x0a)	newline (LF)
'\v'	(0x0b)	vertical tab (VT)
'\f'	(0x0c)	feed (FF)
'\r'	(0x0d)	carriage return (CR)



universität wien

Iterator Invalidierung: Einfügen (Insertion) 1

Container	Rules	n3242 ref.
Sequence containers		
vector	all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)	23.3.6.5/1
deque	all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected)	23.3.3.4/1
list	all iterators and references unaffected	23.3.5.4/1
forward_list	all iterators and references unaffected *(applies to insert_after)*	23.3.4.5/1
array	*(n/a)*	

universität wien

Iterator Invalidierung: Einfügen (Insertion) 2

Associative containers		
set	all iterators and references unaffected	23.2.4/9
multiset		
map		
multimap		
Unordered associative containers		
unordered_set	all iterators invalidated when rehashing occurs, but references unaffected	23.2.5/8
unordered_multiset		
unordered_map		
unordered_multimap		
Container adaptors		
stack	inherited from underlying container	
queue		
priority_queue		

universität wien

Iterator Invalidierung: Löschen (Erasure) 1

Container	Rules	n3242 ref.
Sequence containers		
vector	every iterator and reference after the point of erase is invalidated	23.3.6.5/3
deque	erasing the last element invalidates only iterators and references to the erased elements and the past-the-end iterator; erasing the first element invalidates only iterators and references to the erased elements; erasing any other elements invalidates all iterators and references (including the past-the-end iterator)	23.3.3.4/1
list	only the iterators and references to the erased element is invalidated	23.3.5.4/1
forward_list	only the iterators and references to the erased element is invalidated *(applies to erase_after)*	23.3.4.5/1
array	*(n/a)*	

universität wien

Iterator Invalidierung: Löschen (Erasure) 2

Associative containers		
set	only iterators and references to the erased elements are invalidated	23.2.4/9
multiset		
map		
multimap		
Unordered associative containers		
unordered_set	only iterators and references to the erased elements are invalidated	23.2.5/13
unordered_multiset		
unordered_map		
unordered_multimap		
Container adaptors		
stack	inherited from underlying container	
queue		
priority_queue		

universität wien

Iterator Invalidierung: Größenänderung (Resizing)

Container	Rules	n3242 ref.
Sequence containers		
vector	as per insert/erase	23.3.6.5/12
deque	as per insert/erase	23.3.3.4/1
list	as per insert/erase	23.3.5.3/1
forward_list	as per insert/erase	23.3.4.5/25
array	*(n/a)*	

universität wien

Iterator Invalidierung: Bemerkungen

Footnote 1

Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container. [23.2.1/11]

Footnote 2

No swap() function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: The end() iterator does not refer to any element, so it may be invalidated. —end note] [23.2.1/10]

Footnote 3

Other than the above caveat regarding swap() and the erasure rule for deque, it's not clear whether "end" iterators are subject to the above listed per-container rules; you should assume, anyway, that they are.

Quelle: http://kera.name/articles/2011/06/iterator-invalidation-rules-c0x/

universität wien

Anhang: Trennzeichen (white-space characters)

In der Regel werden die folgenden Zeichen als Trennzeichen betrachtet. (Das kann aber durch länderspezifische Einstellungen sogenannte locale settings modifiziert werden.)

' '	(0x20)	space (SPC)
'\t'	(0x09)	horizontal tab (TAB)
'\n'	(0x0a)	newline (LF)
'\v'	(0x0b)	vertical tab (VT)
'\f'	(0x0c)	feed (FF)
'\r'	(0x0d)	carriage return (CR)

universität wien

Dynamische Folien

universität wien

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
```

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
```

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
0 ist gar nichts
```

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
0 ist gar nichts
0.5 ist auch nicht viel, aber immerhin
```

Beispiel: Datentypen (1)

```
#include<iostream>
using namespace std;
int main()

        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
0 ist gar nichts
0.5 ist auch nicht viel, aber immerhin
1 ist auch ohne Komma eine reelle Zahl
```

Beispiel: Date

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

Beispiel: Date

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
0 ist gar nichts
0.5 ist auch nicht viel, aber immerhin
1 ist auch ohne Komma eine reelle Zahl
i ist gleich 1 und ist ungerade
```

```
#include<iostream>
using namespace std;

int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

Beispiel: Date

```
Pi ist gleich 3.14159
Der Umfang des Kreises mit Radius 1.2 betraegt 7.53982
7 Tage hat die Woche
0 ist gar nichts
0.5 ist auch nicht viel, aber immerhin
1 ist auch ohne Komma eine reelle Zahl
i ist gleich 1 und ist ungerade
Viel Spass mit C++
```

```
#include<iostream>
using namespace std;

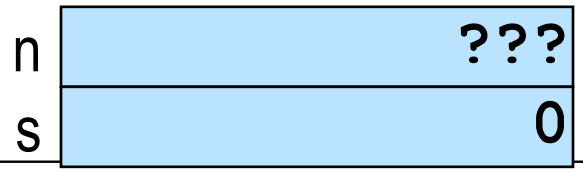
int main()
{
        int i {1}, j {2};
        constexpr double pi {3.14159};
        double r {1.2}, U;
        // Allgemein wird der "maechtigere" Datentyp fuer das Ergebnis
        // des Ausdrucks verwendet
        cout << "Pi ist gleich " << i*pi << '\n';
        U = 2*r*pi;
        cout << "Der Umfang des Kreises mit Radius " << r << " betraegt " << U << '\n';
        // Aber bei Zuweisung wird bei Bedarf abgeschnitten
        j = U;
        cout << j << " Tage hat die Woche\n";
        // Die Division liefert ein ganzzahliges Ergebnis, wenn beide Operanden
        // ganzzahlig sind.
        cout << i/2 << " ist gar nichts\n";
        r = i; // r bleibt trotzdem eine reelle Zahl!
        cout << r/2 << " ist auch nicht viel, aber immerhin\n";
        cout << r << " ist auch ohne Komma eine reelle Zahl\n";
        // Der Operator % ist (in C++) nur fuer ganzzahlige Operanden definiert
        cout << "i" << " ist gleich " << i << " und ";
        if ((i % 2) == 0) cout << "ist gerade\n";
        else cout << "ist ungerade\n";
        // r % 2; ist verboten
        char first {'C'};
        char rest[3] {"++"}; // unterschiedliche Hochkommata und Laenge 3 beachten
        cout << "Viel Spass mit " << first << rest << '\n';
        return 0;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```



universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

false

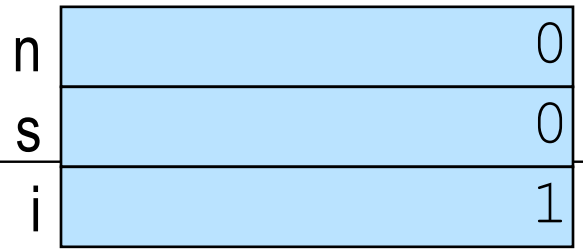
universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

Ausgabe: 0 (das ist korrekt)

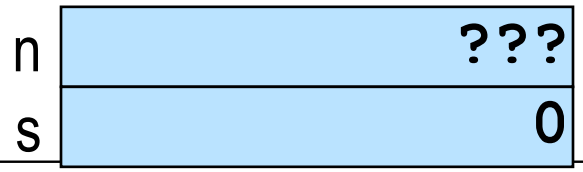


universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```



universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

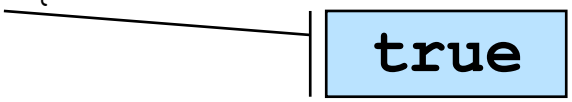
using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```



universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

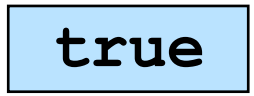
using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```



universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

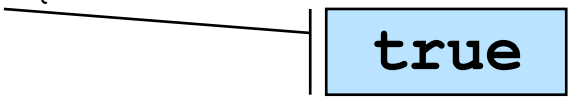
using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```



universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

false

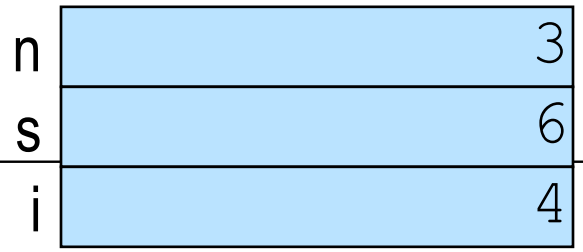
universität wien

while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  while (i <= n) {
    s = s + i;
    i = i + 1;
  }
  cout << s;
}
```

Ausgabe: 6

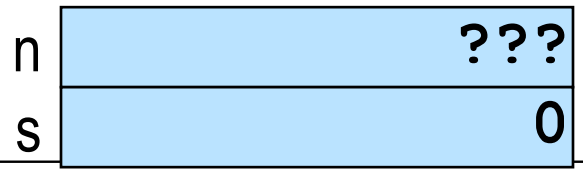


universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```



universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

false

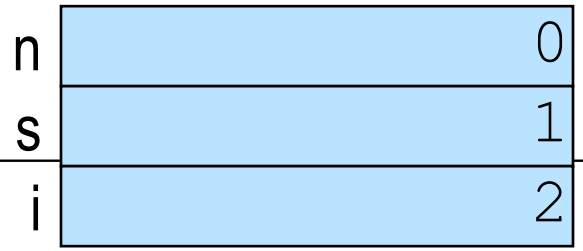
universität wien

do while

```
#include<iostream>

using namespace std;
int main() {
  int n, s {0};
  int i {1};
  cin >> n;
  do {
    s = s + i;
    i = i + 1;
  } while (i <= n);
  cout << s;
}
```

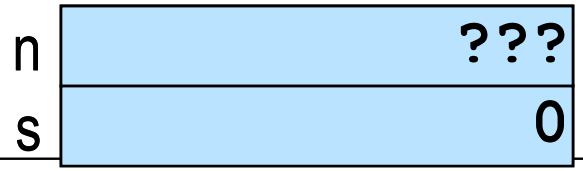
Ausgabe: 1 (das ist falsch!)



universität wien

for

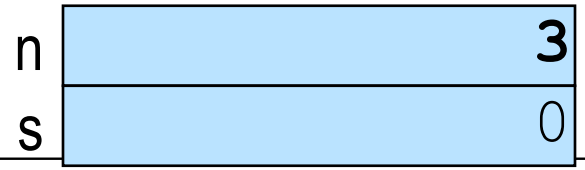
```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```



universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```



universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

true

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

true

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

true

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

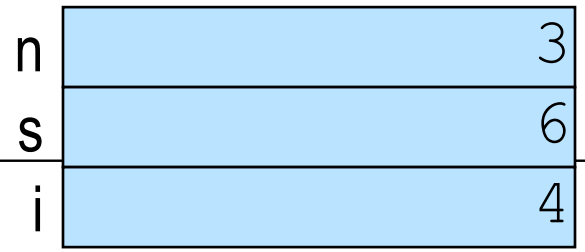
```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

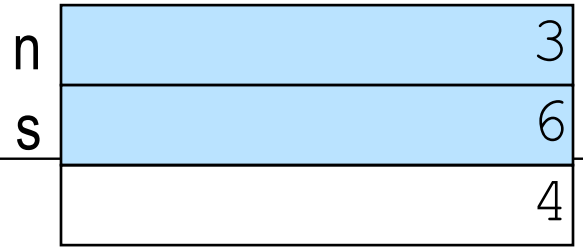
false



universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

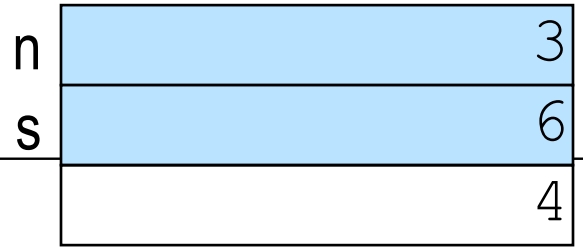


universität wien

for

```
#include<iostream>
using namespace std;
int main () {
  int n, s {0};
  cin >> n;
  for (int i {1}; i <= n; ++i) {
    s = s + i;
  }
  cout << s;
}
```

Ausgabe: 6



universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

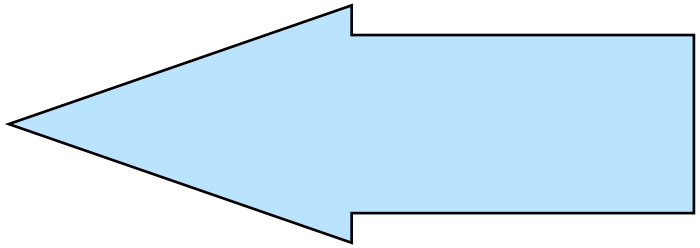
Programmablauf beginnt hier

Ausgabe:

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

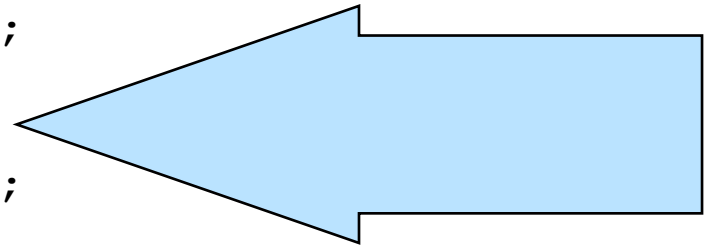


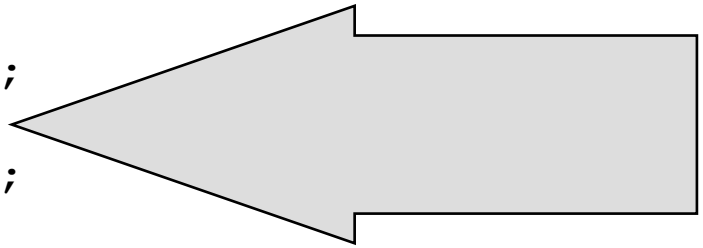
Ausgabe:
4

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```



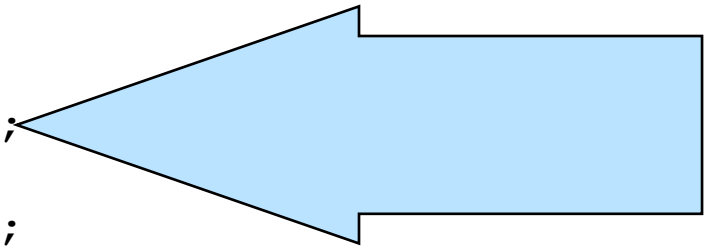


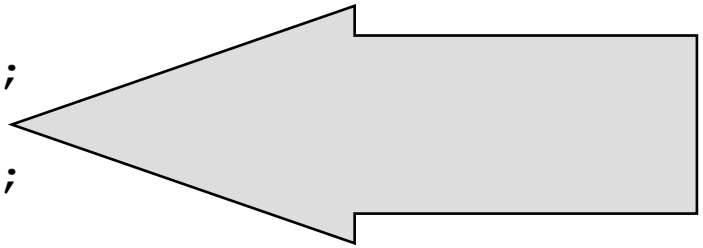
Ausgabe:
4

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```



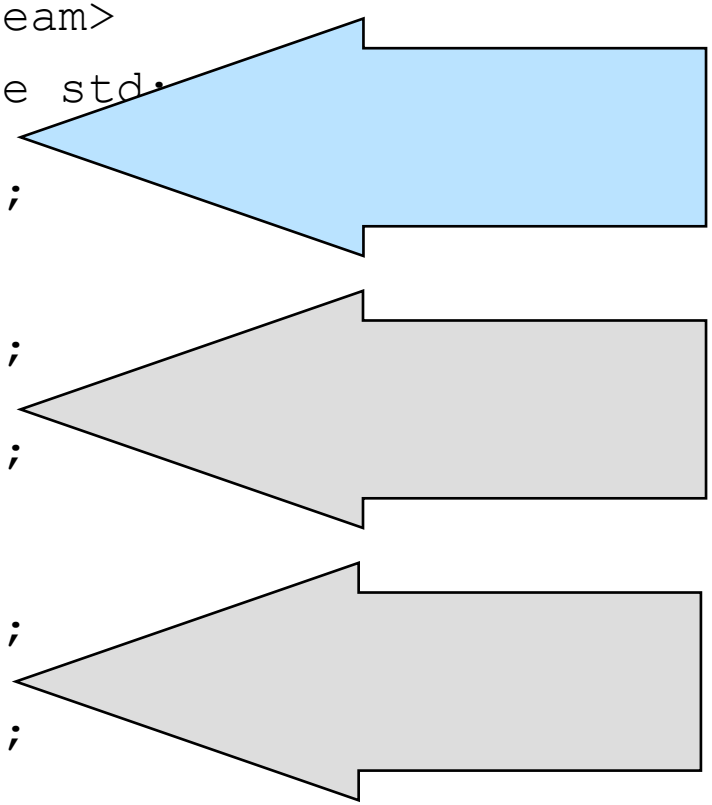


Ausgabe:
4 2

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

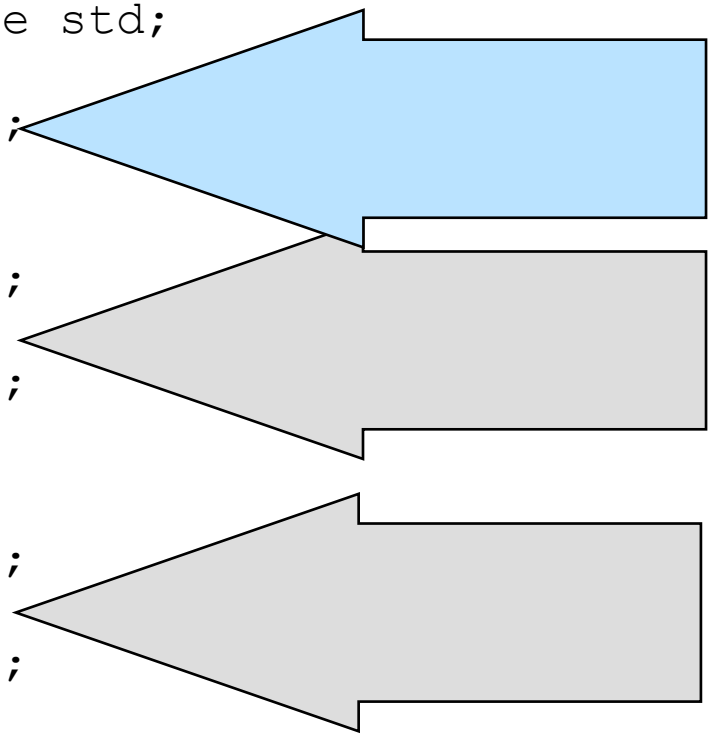


Ausgabe:
4 2

universität wien

Funktionen (2)

- `#include<iostream>`
-
```
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
```
-
```
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

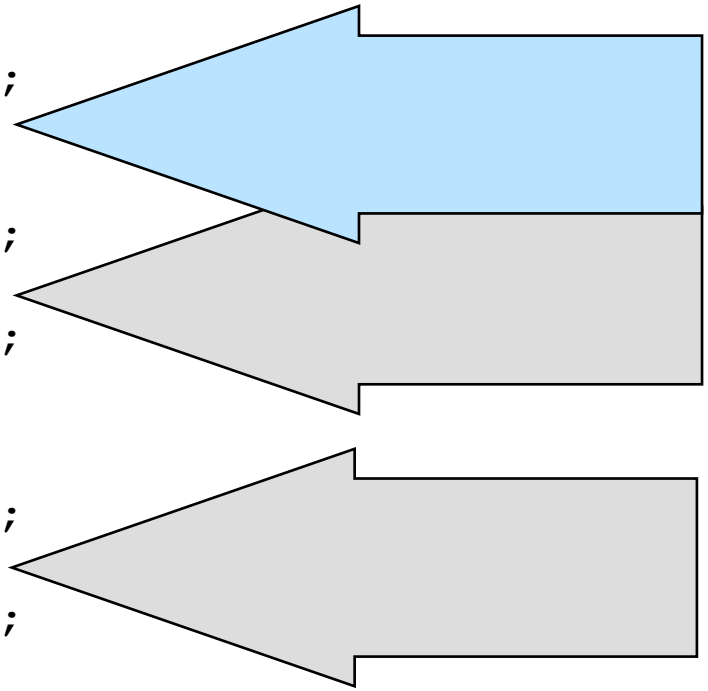


Ausgabe:
4 2 1

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

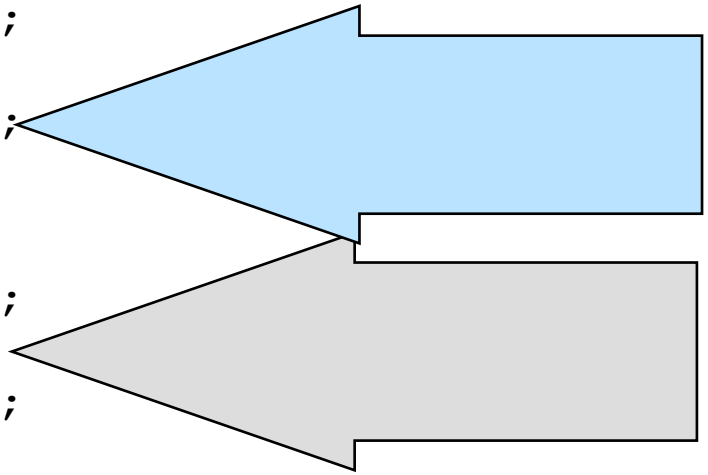


Ausgabe:
4 2 1

universität wien

Funktionen (2)

- `#include<iostream>`
- ```
  using namespace std;
  void b() {
    cout << "1 ";
  }
  void a() {
    cout << "2 ";
    b();
    cout << "3 ";
  }
  ```
- ```
  int main() {
    cout << "4 ";
    a();
    cout << "5 ";
    return 0;
  }
  ```

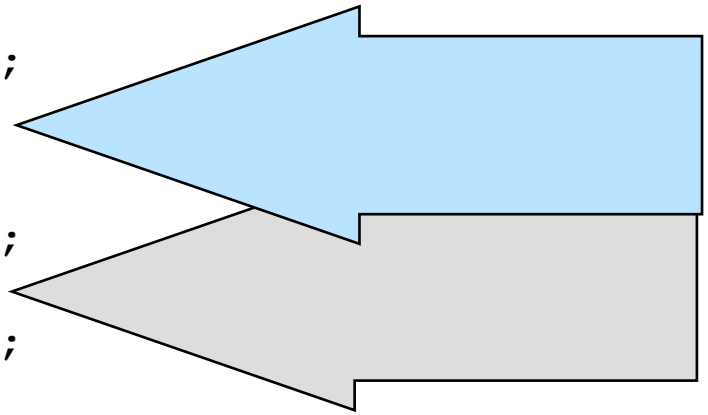


Ausgabe:
4 2 1 3

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

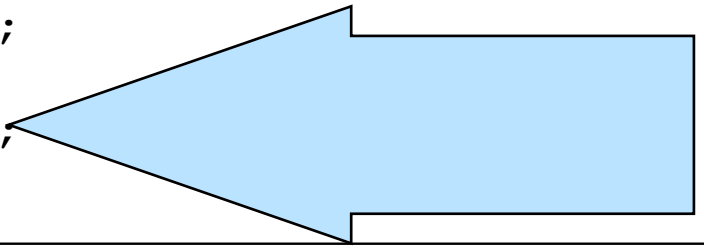


Ausgabe:
4 2 1 3

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```



Ausgabe:
4 2 1 3 5

universität wien

Funktionen (2)

```
#include<iostream>
using namespace std;
void b() {
  cout << "1 ";
}
void a() {
  cout << "2 ";
  b();
  cout << "3 ";
}
int main() {
  cout << "4 ";
  a();
  cout << "5 ";
  return 0;
}
```

Ausgabe:
4 2 1 3 5

Rückgabe

universität wien

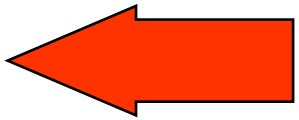
Funktionsaufrufmechanismus

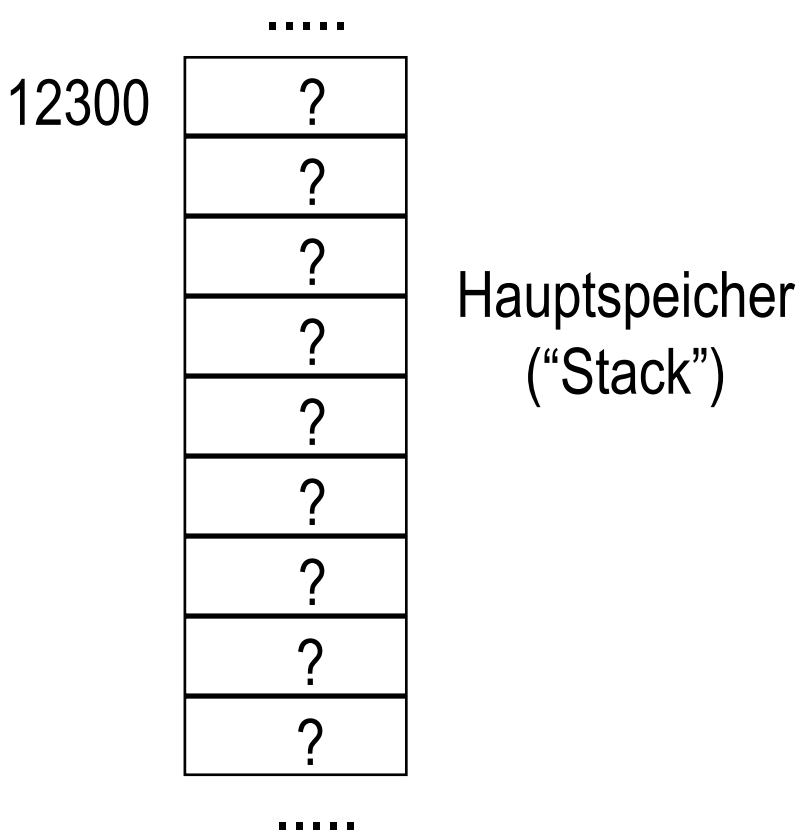
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);

  return 0;
}
```





universität wien

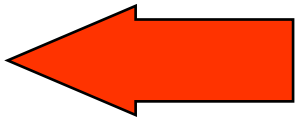
Funktionsaufrufmechanismus

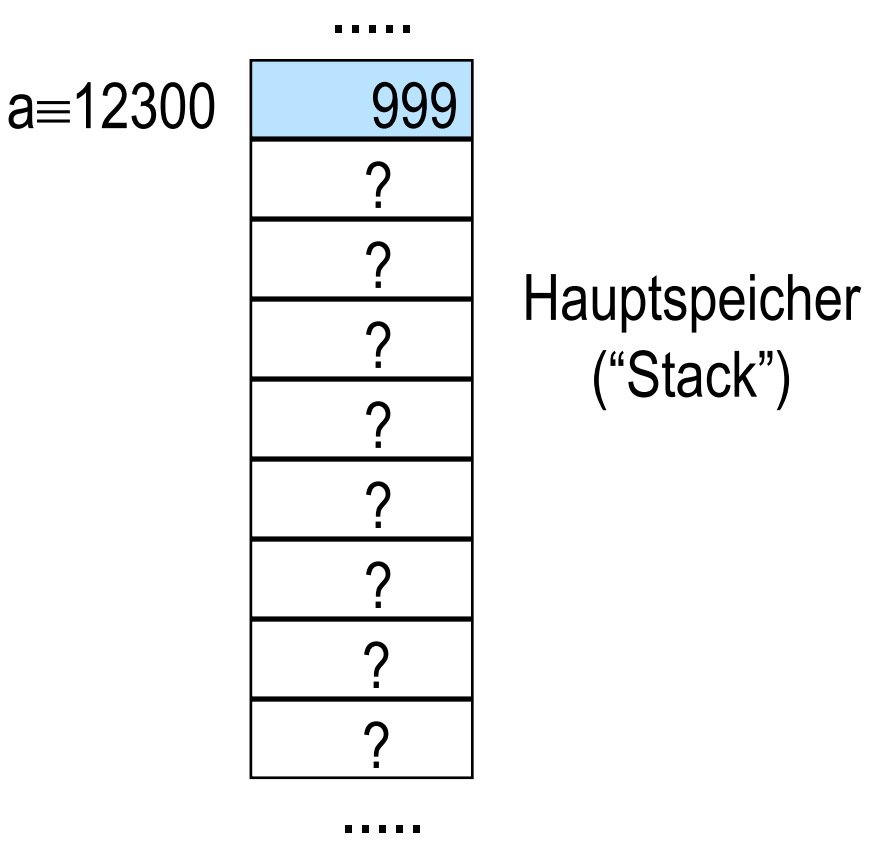
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);

  return 0;
}
```





universität wien

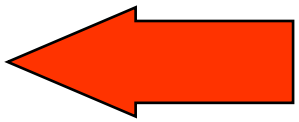
Funktionsaufrufmechanismus

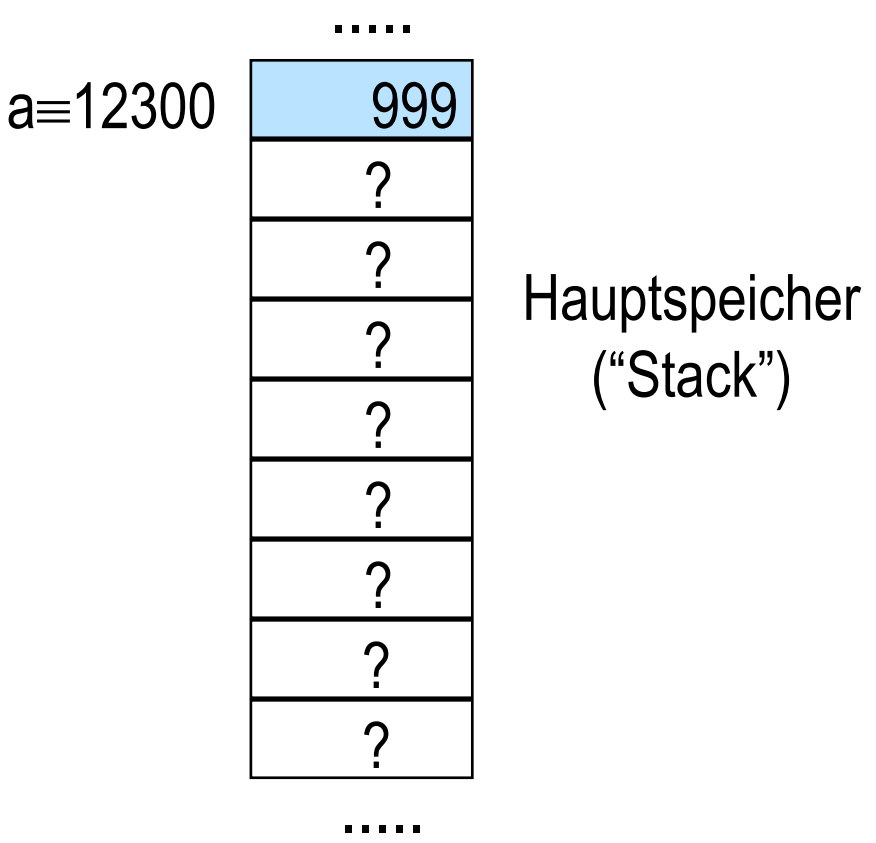
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);

  return 0;
}
```





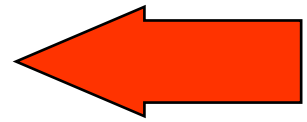
universität wien

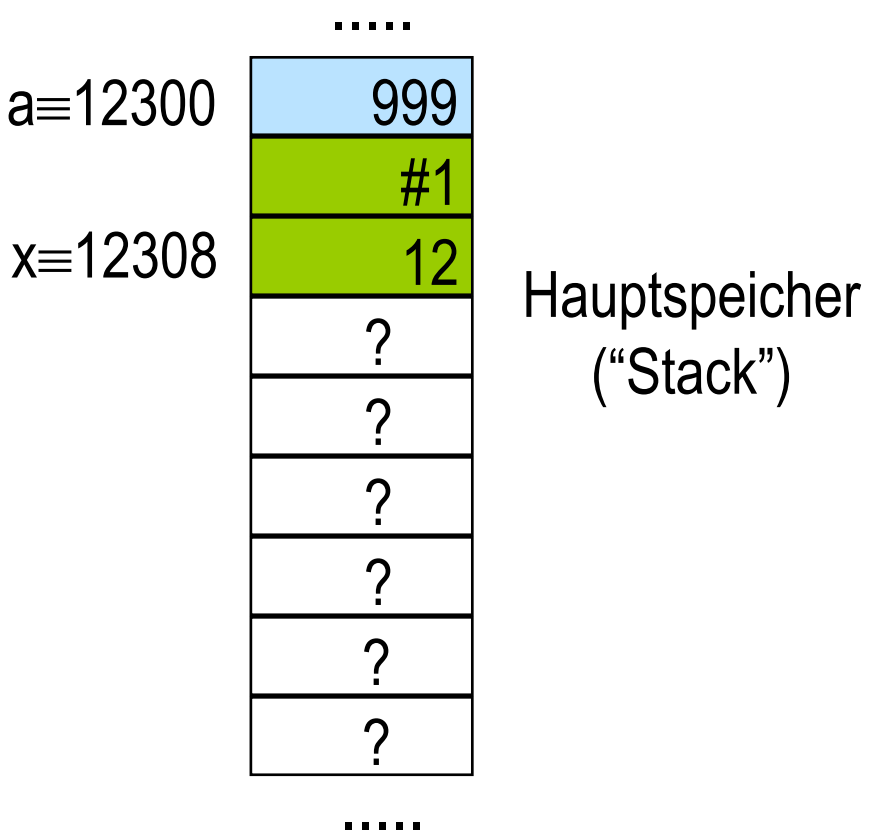
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```





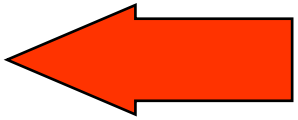
universität wien

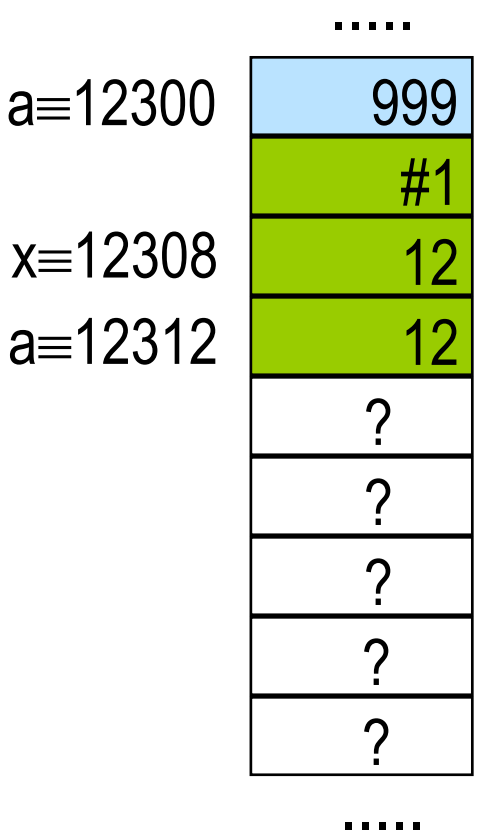
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```





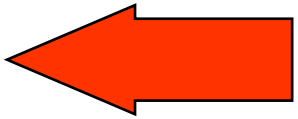
universität wien

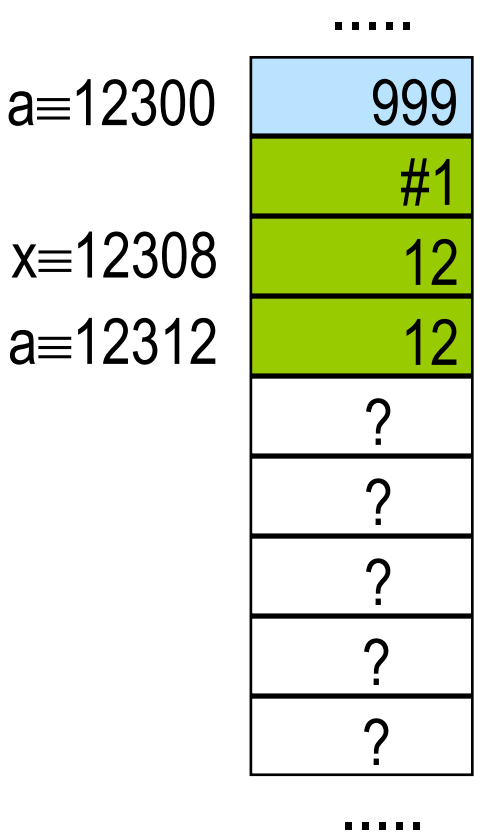
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```





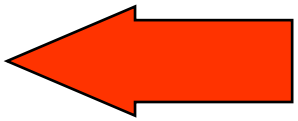
universität wien

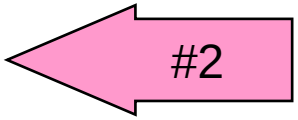
Funktionsaufrufmechanismus

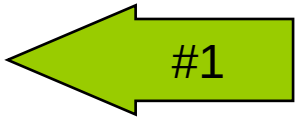
```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

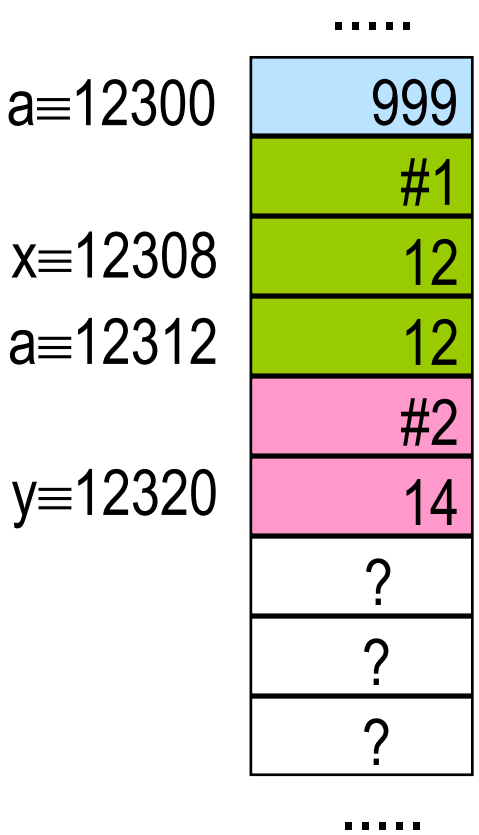
```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```









Hauptspeicher
("Stack")

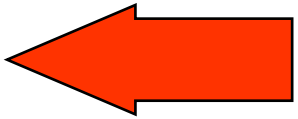
universität wien

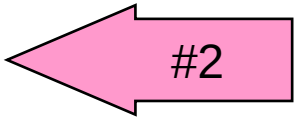
Funktionsaufrufmechanismus

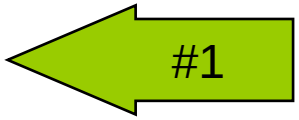
```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

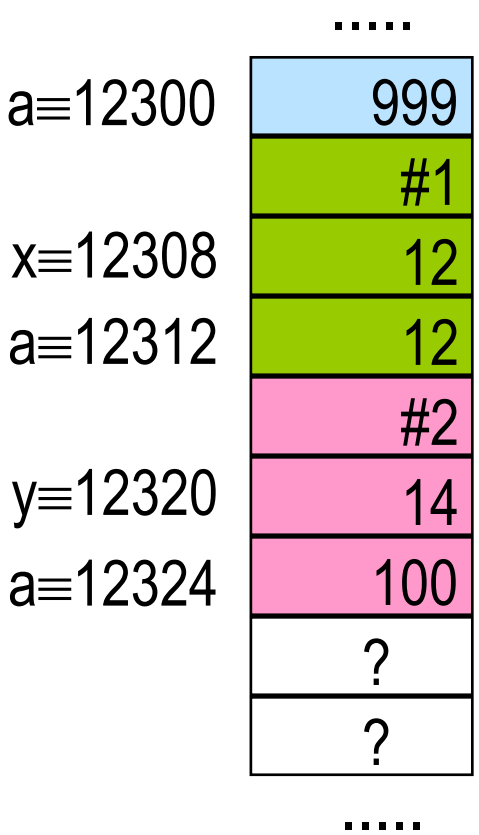
```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```









Hauptspeicher
("Stack")

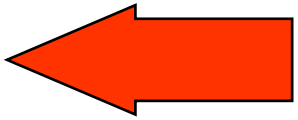
universität wien

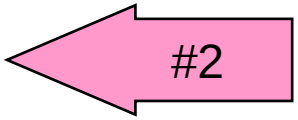
Funktionsaufrufmechanismus

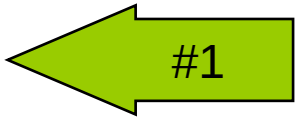
```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

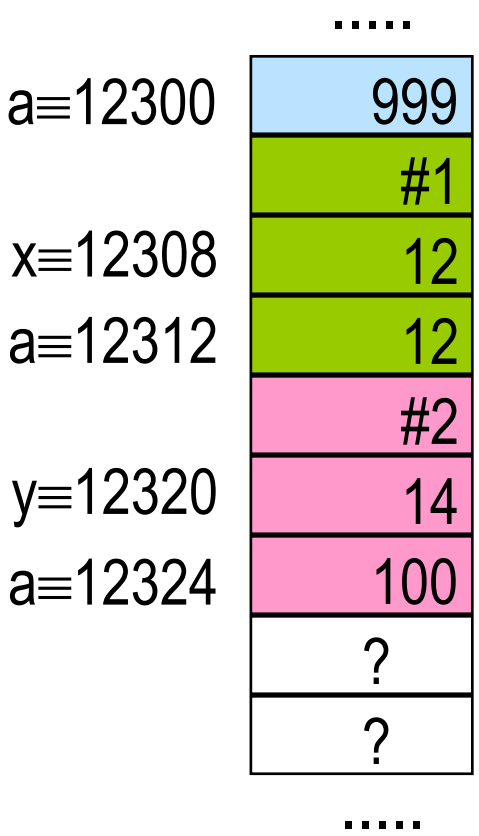
```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```









Hauptspeicher
("Stack")

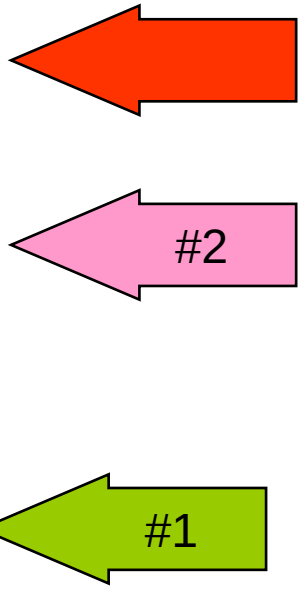
universität wien

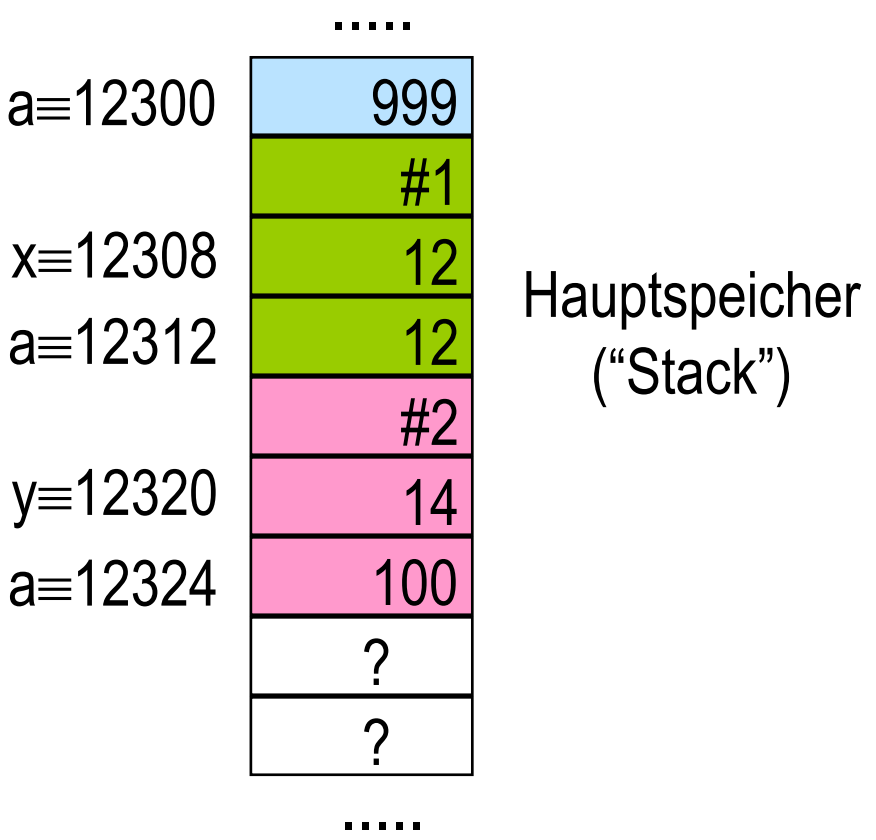
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```





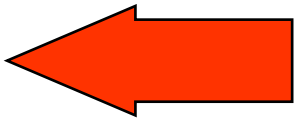
universität wien

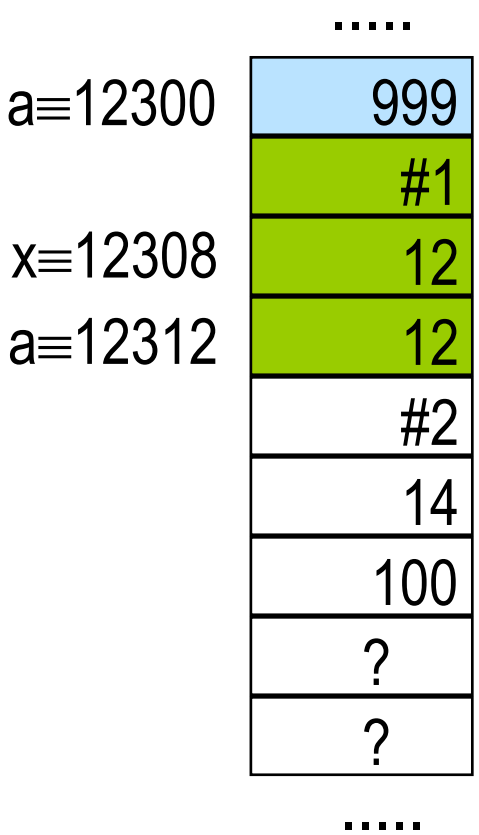
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```





11412

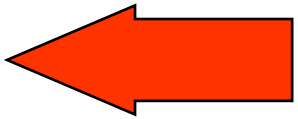
universität wien

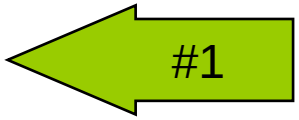
Funktionsaufrufmechanismus

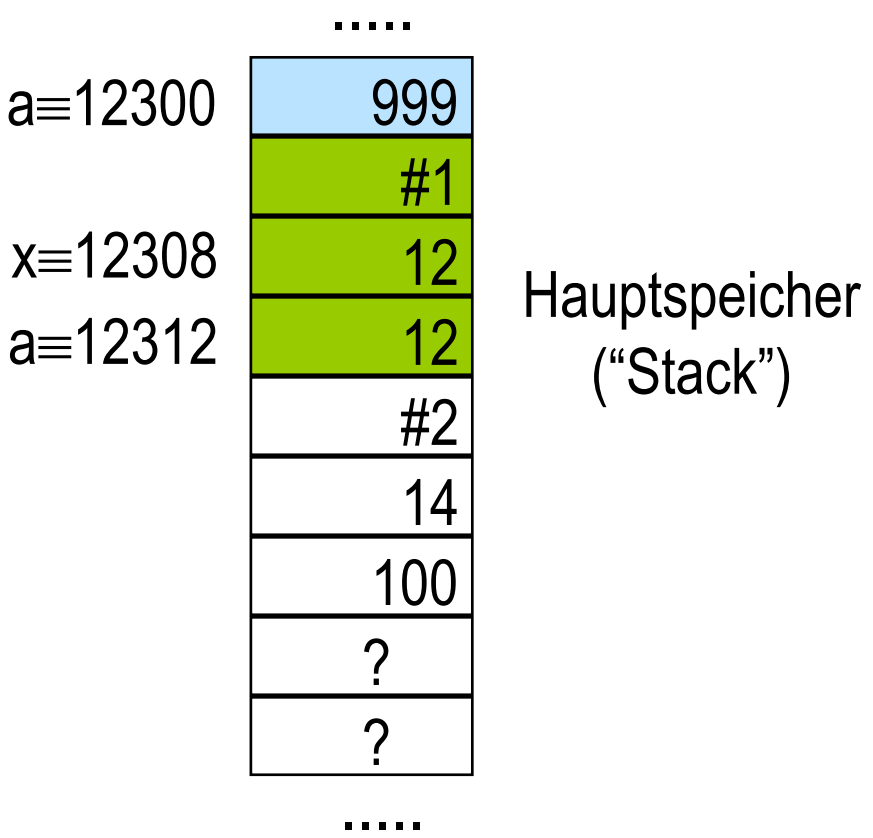
```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```







11412

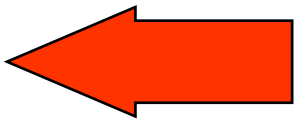
Funktionsaufrufmechanismus

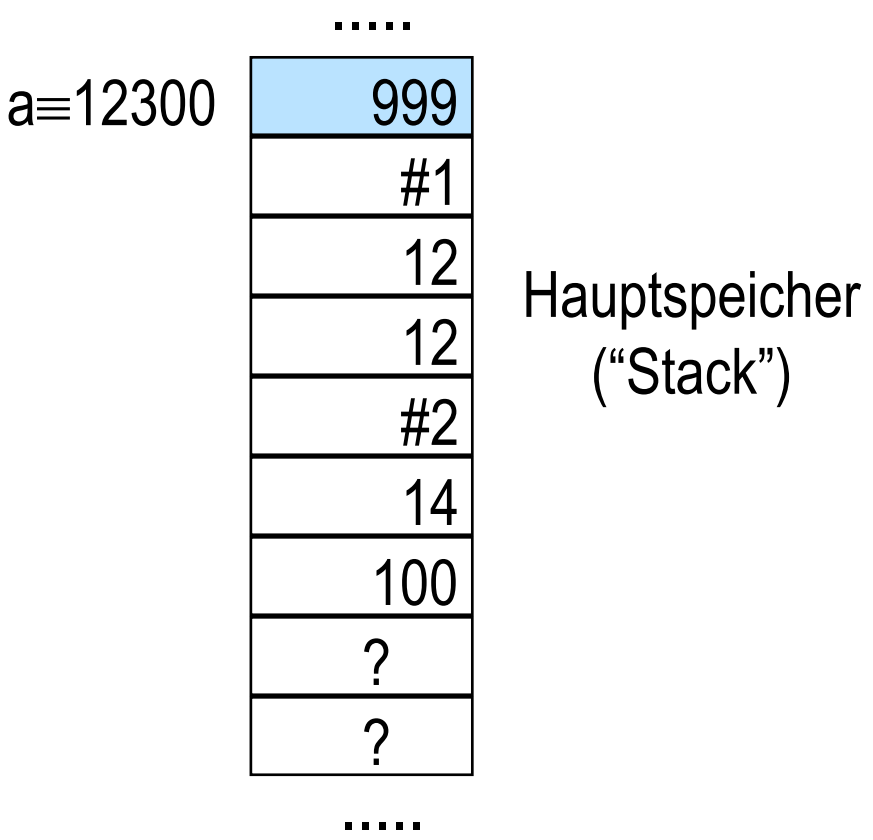
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);

  return 0;
}
```





11412

universität wien

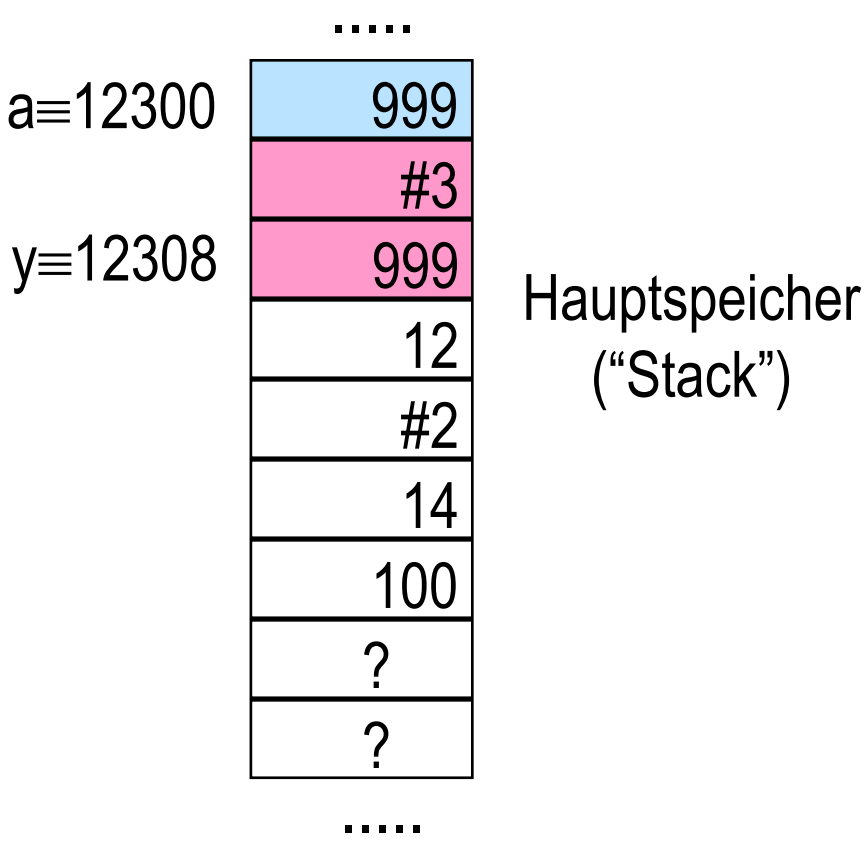
Funktionsaufrufmechanismus

```
void g (int y) {
  int a {100};
  cout << y+a;
}
```

```
void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}
```

```
int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```

#3



11412

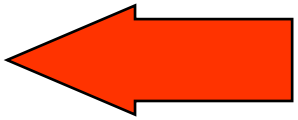
universität wien

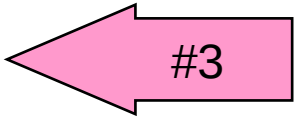
Funktionsaufrufmechanismus

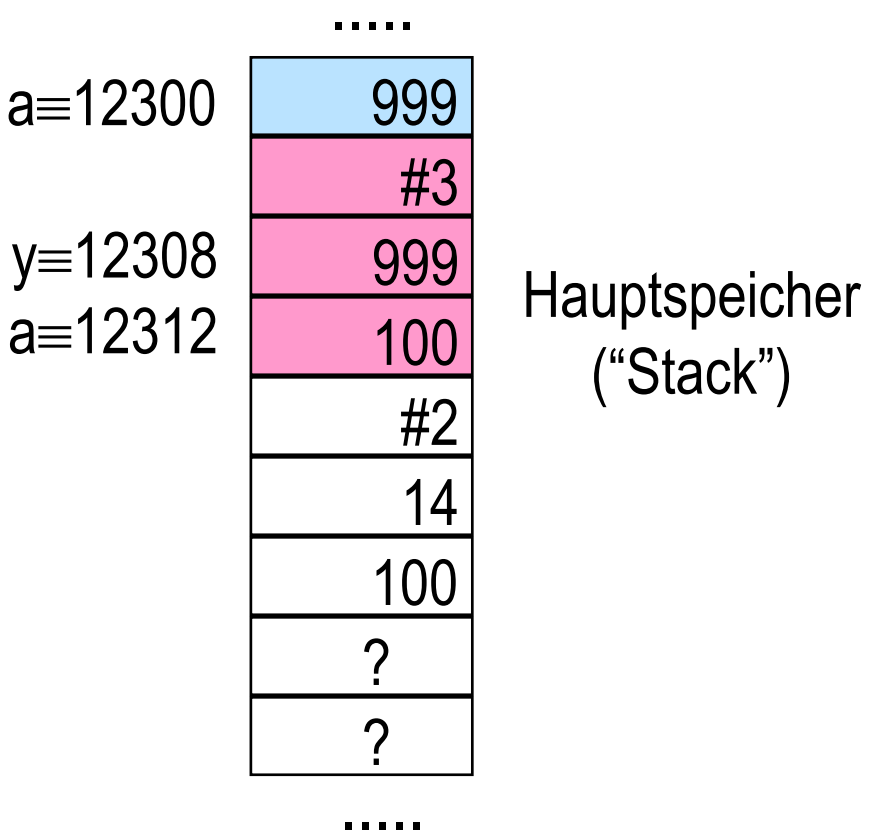
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```







11412

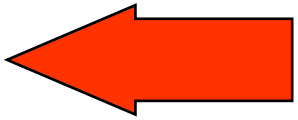
universität wien

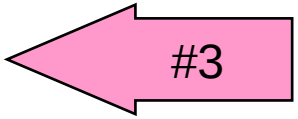
Funktionsaufrufmechanismus

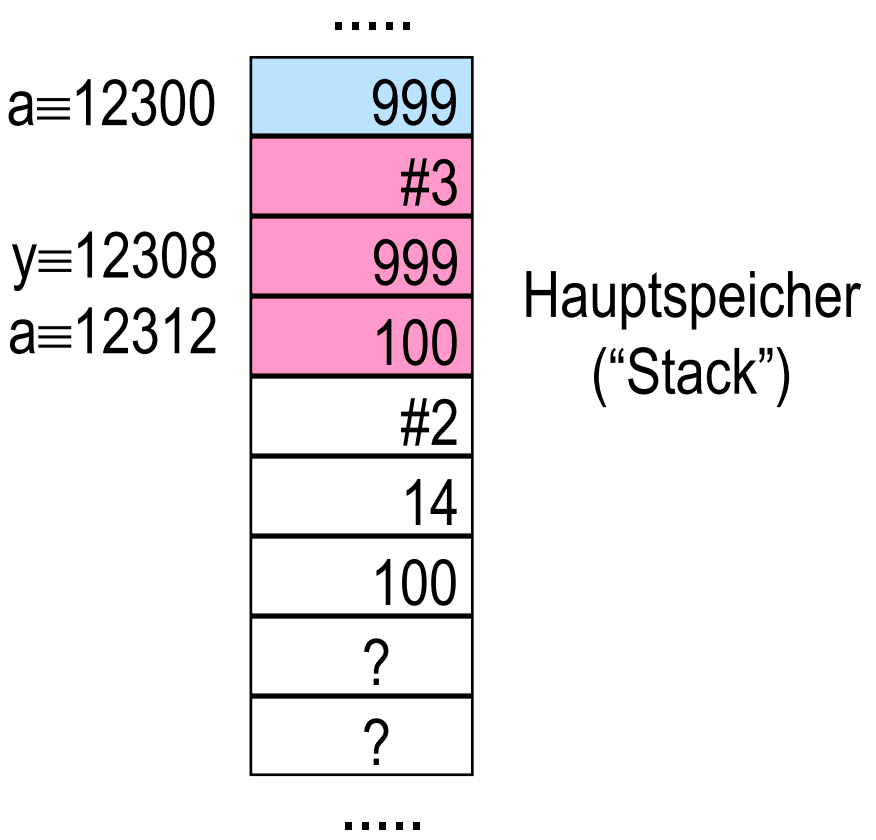
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```







114121099

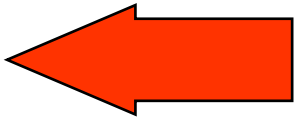
universität wien

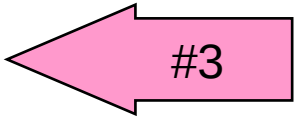
Funktionsaufrufmechanismus

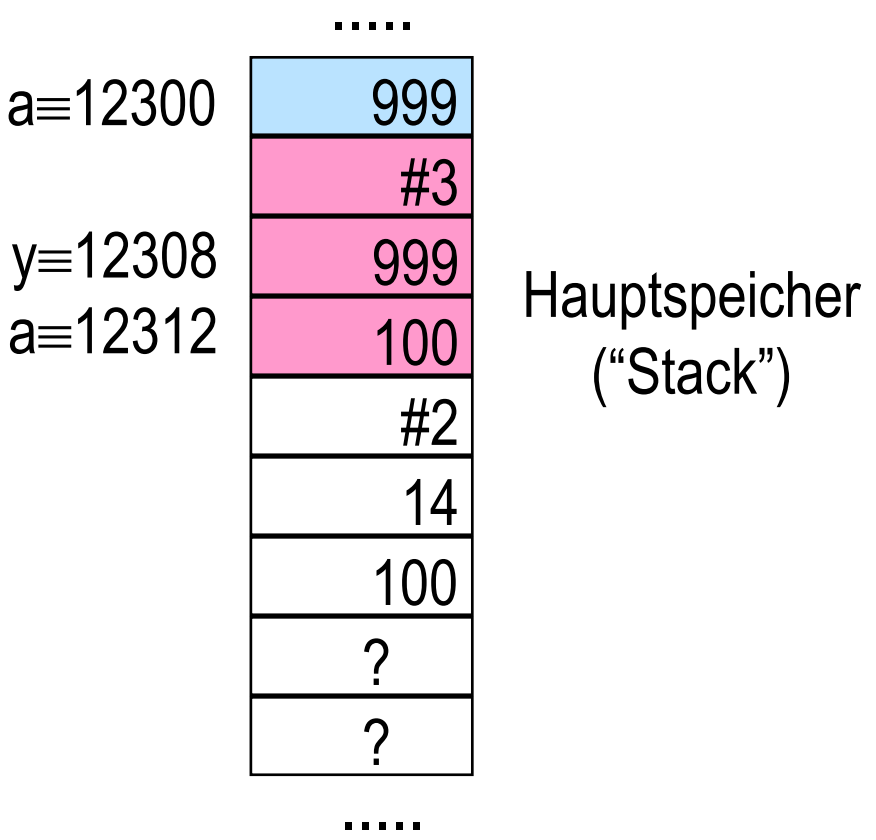
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);
  return 0;
}
```







114121099

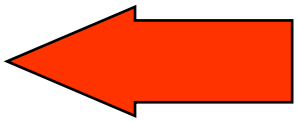
Funktionsaufrufmechanismus

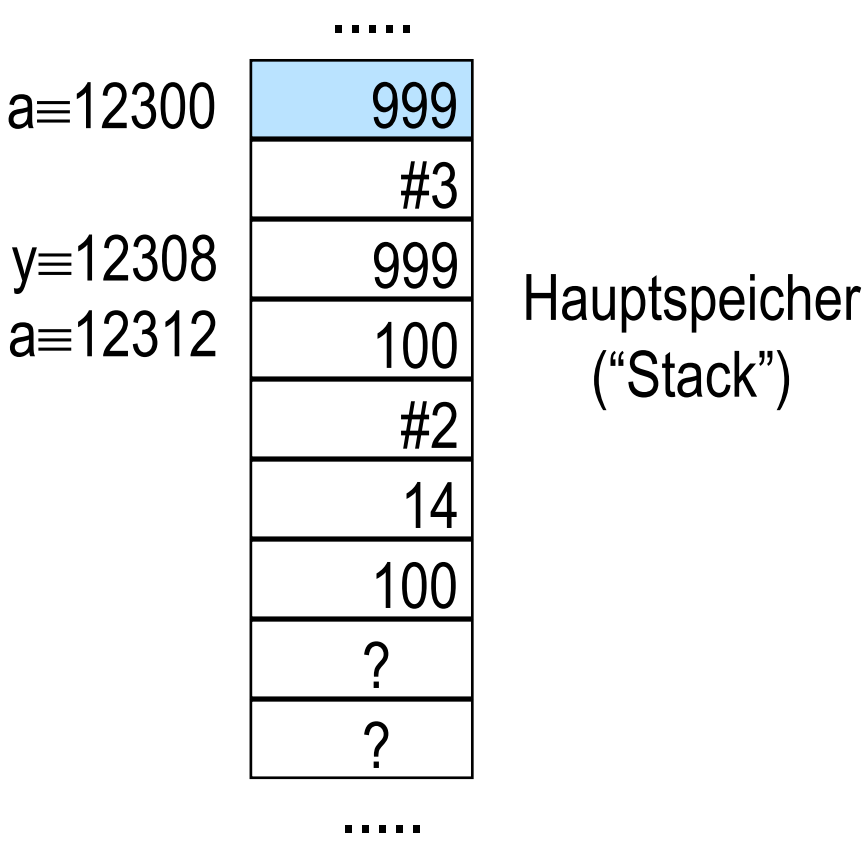
```
void g (int y) {
  int a {100};
  cout << y+a;
}

void f (int x) {
  int a {x};
  g(a+2);
  cout << a;
}

int main () {
  int a {999};
  f(12);
  g(a);

  return 0;
}
```





114121099

universität wien

Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```

universität wien

Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

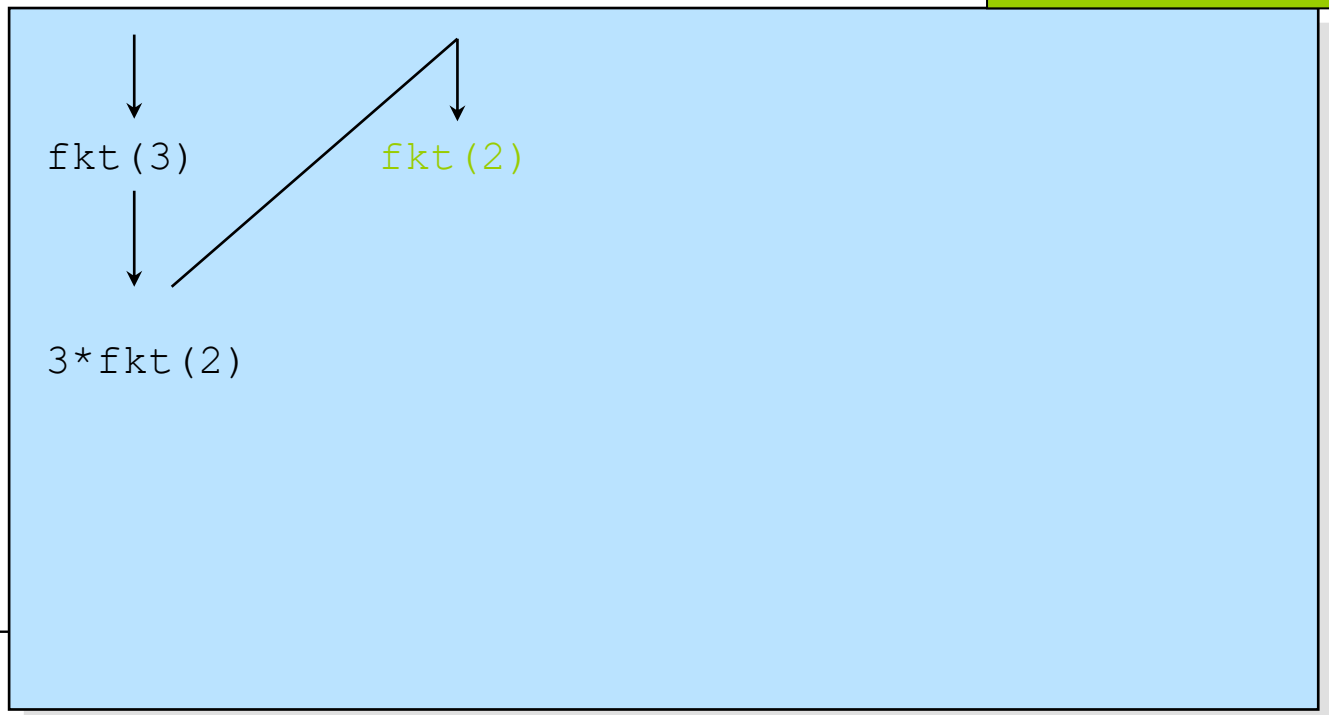
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```

universität wien

Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

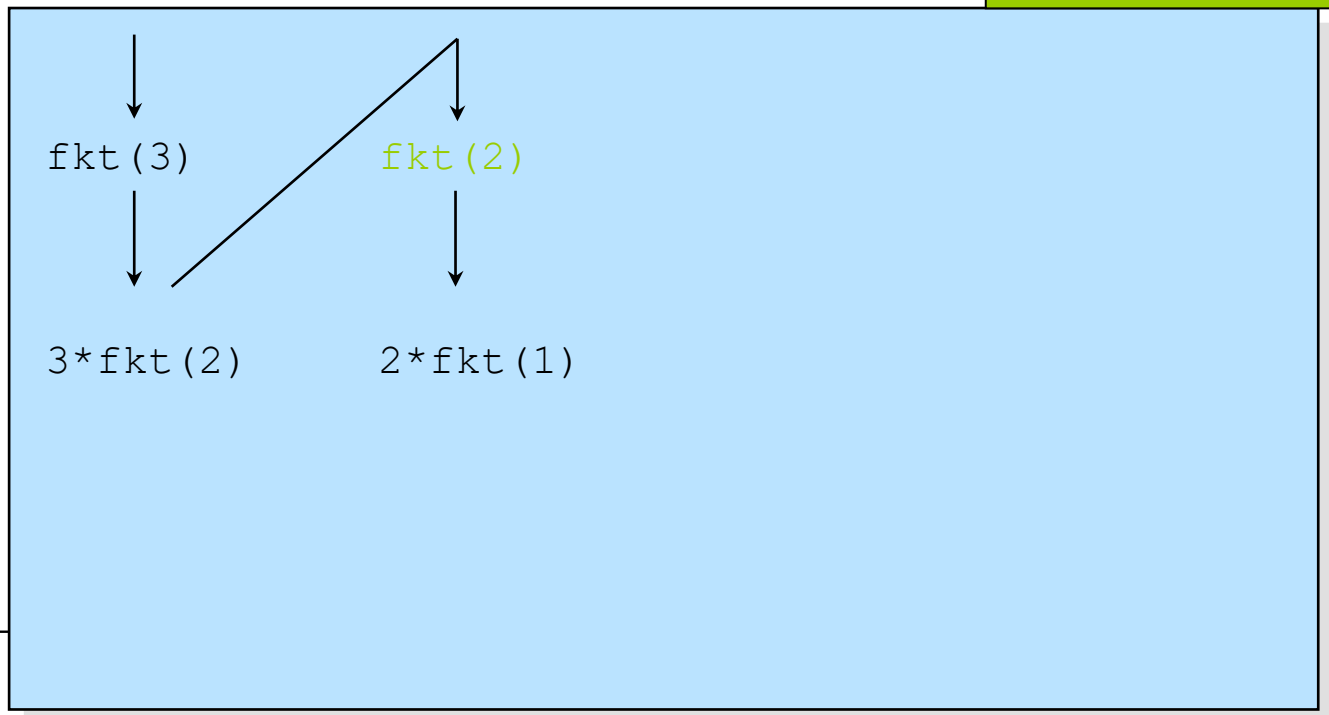
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

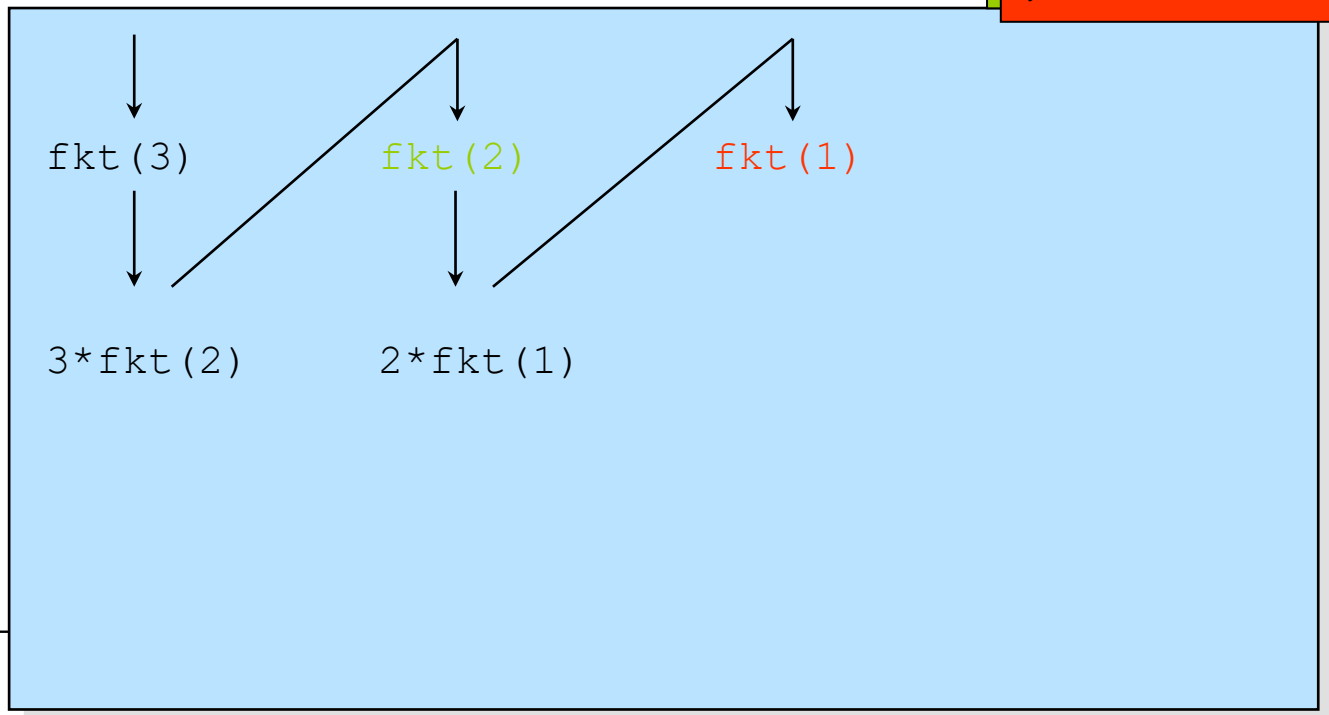
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```

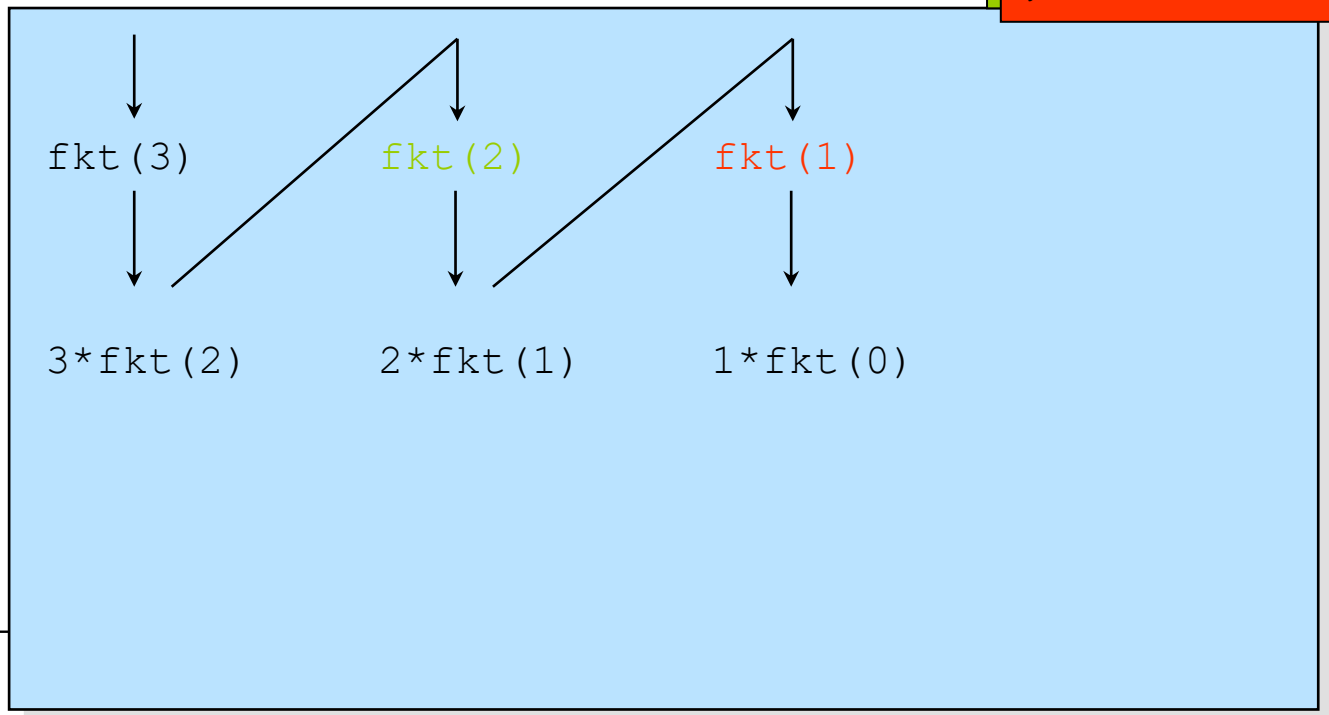


universität wien

Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

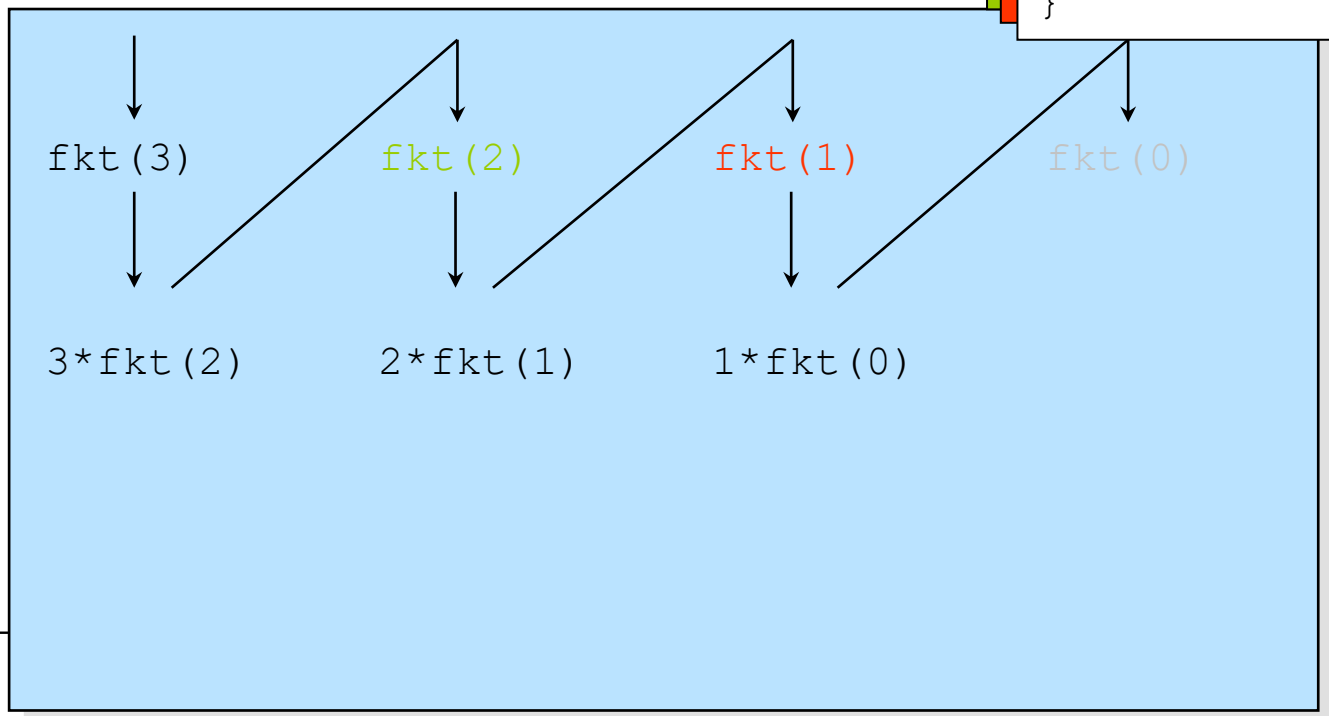
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

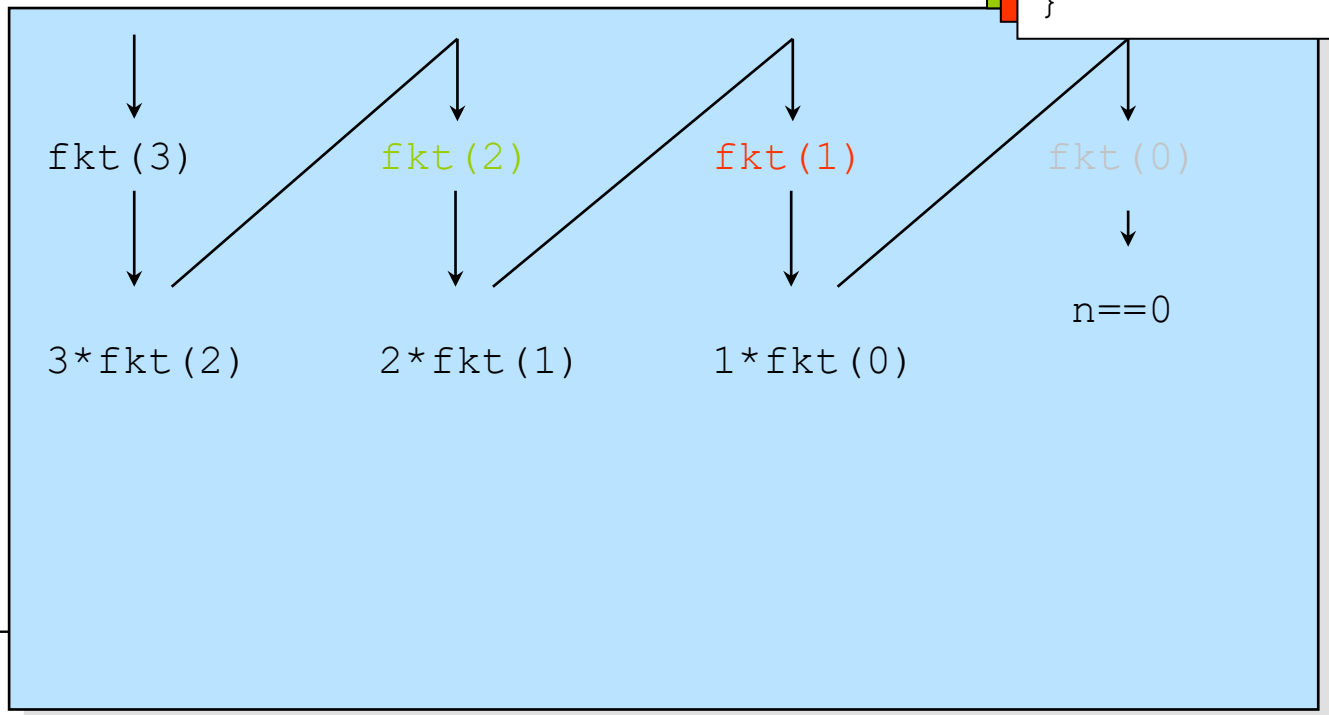
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

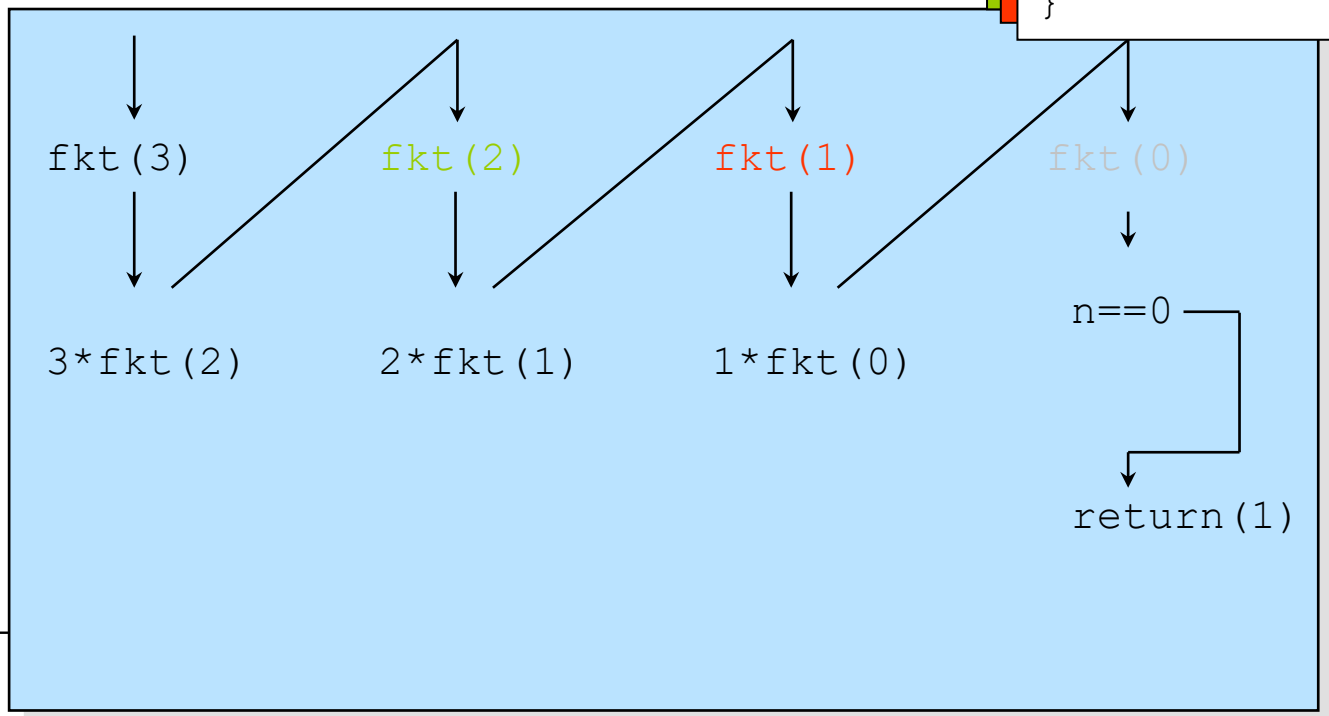
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

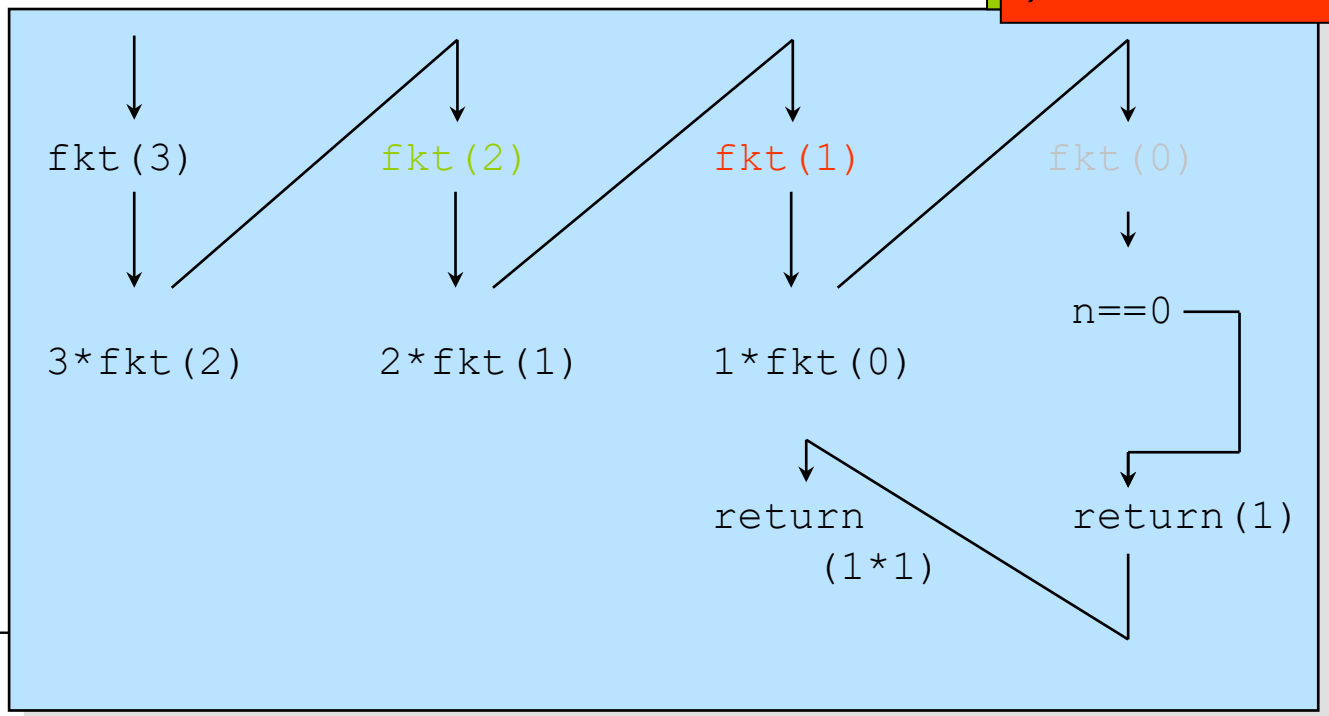
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

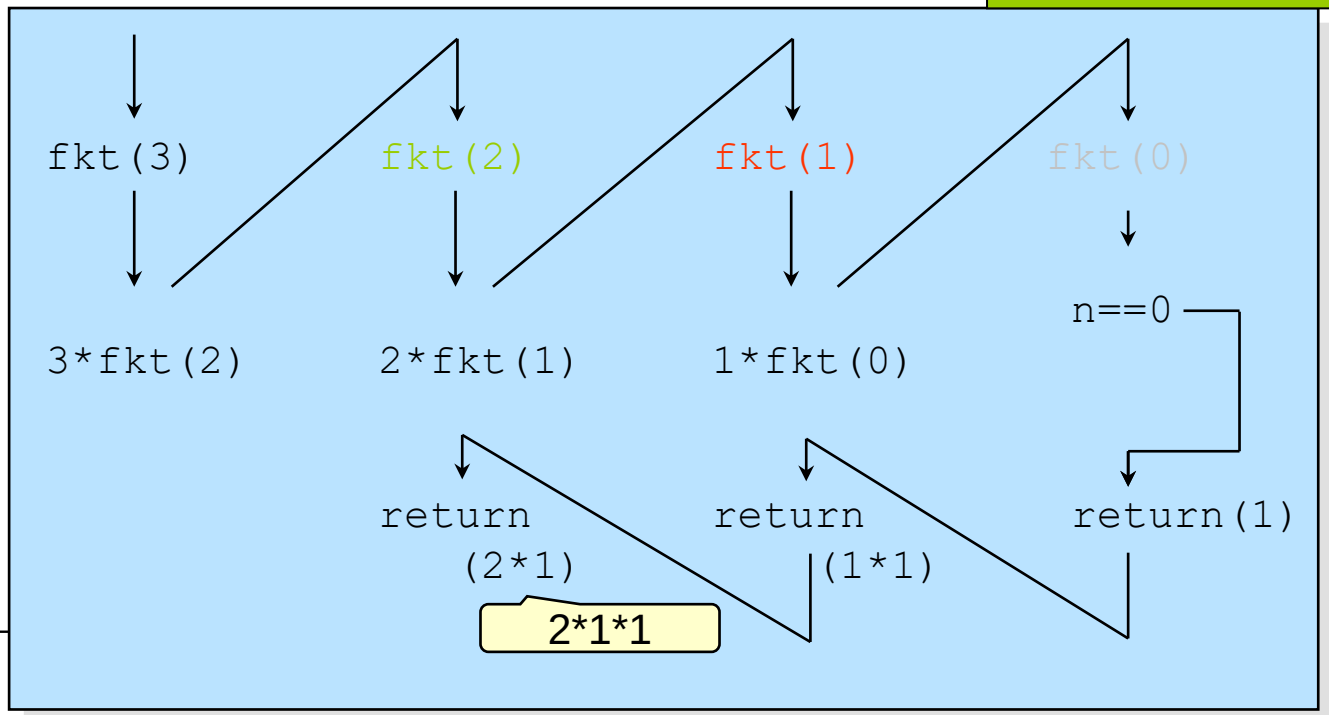
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

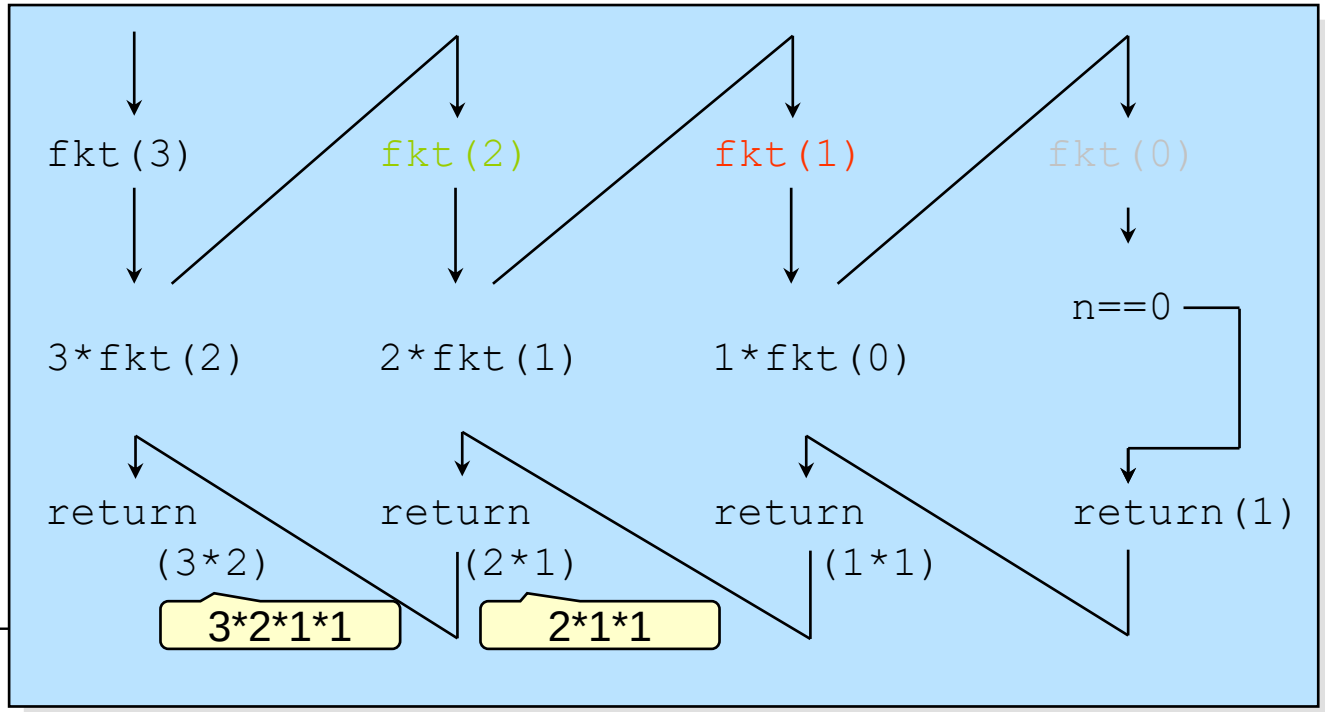
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```

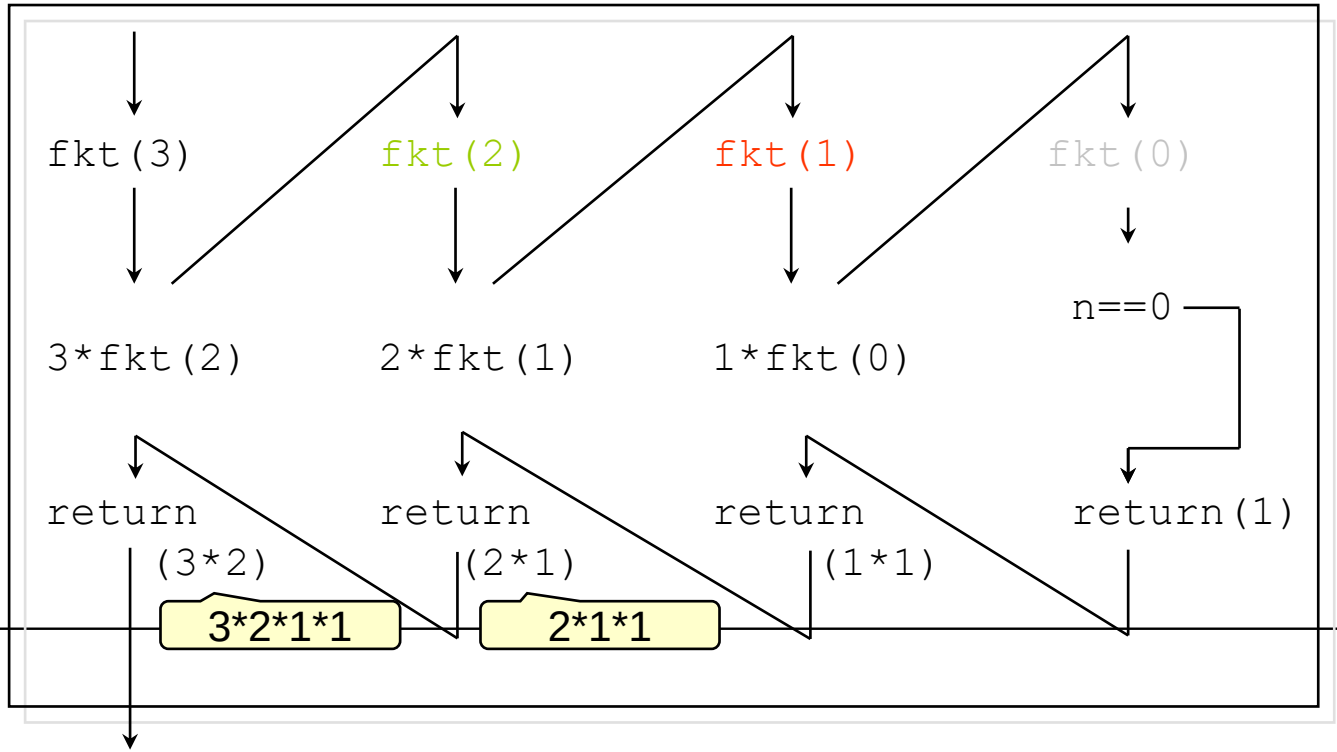


universität wien

Direkte Rekursion: Fakultät

- Berechnung von 3!
  - Aufruf: fkt(3)

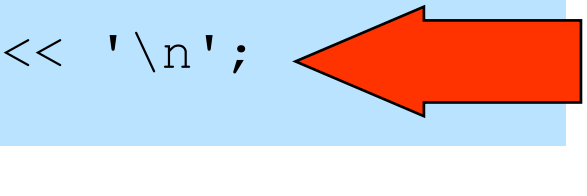
```
int fkt(int n) {
  if(n == 0)
    return(1);
  else
    return(n*fkt(n-1));
}
```



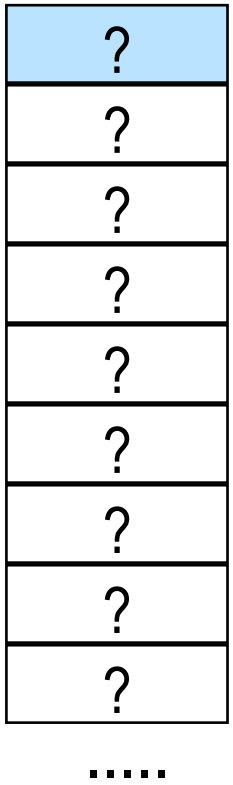
universität wien

Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```



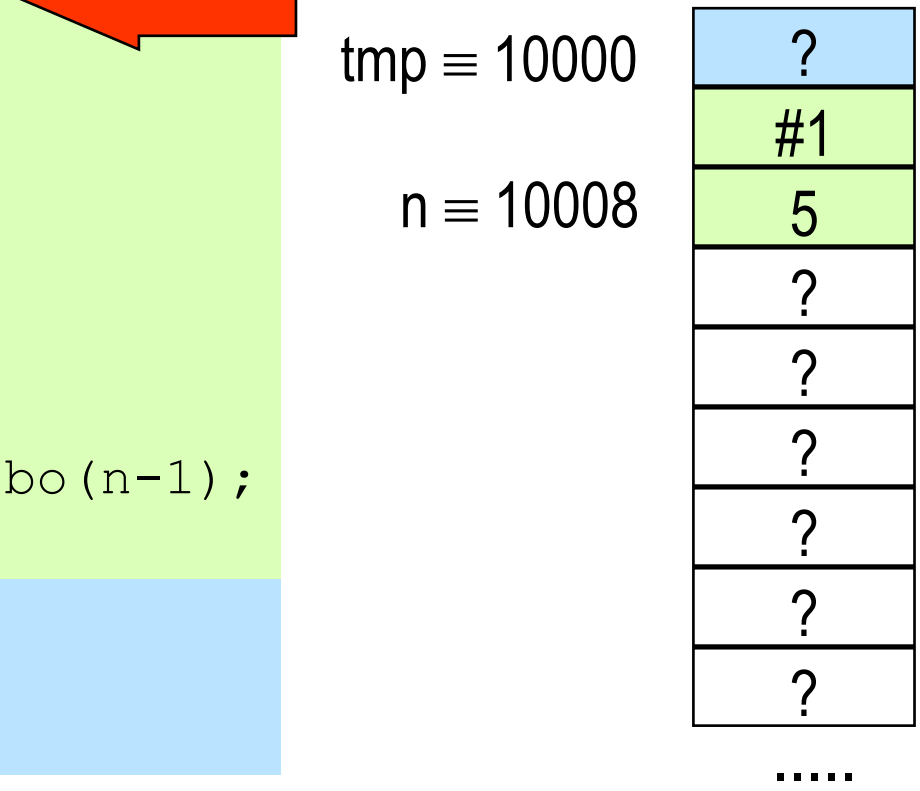
tmp ≡ 10000



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

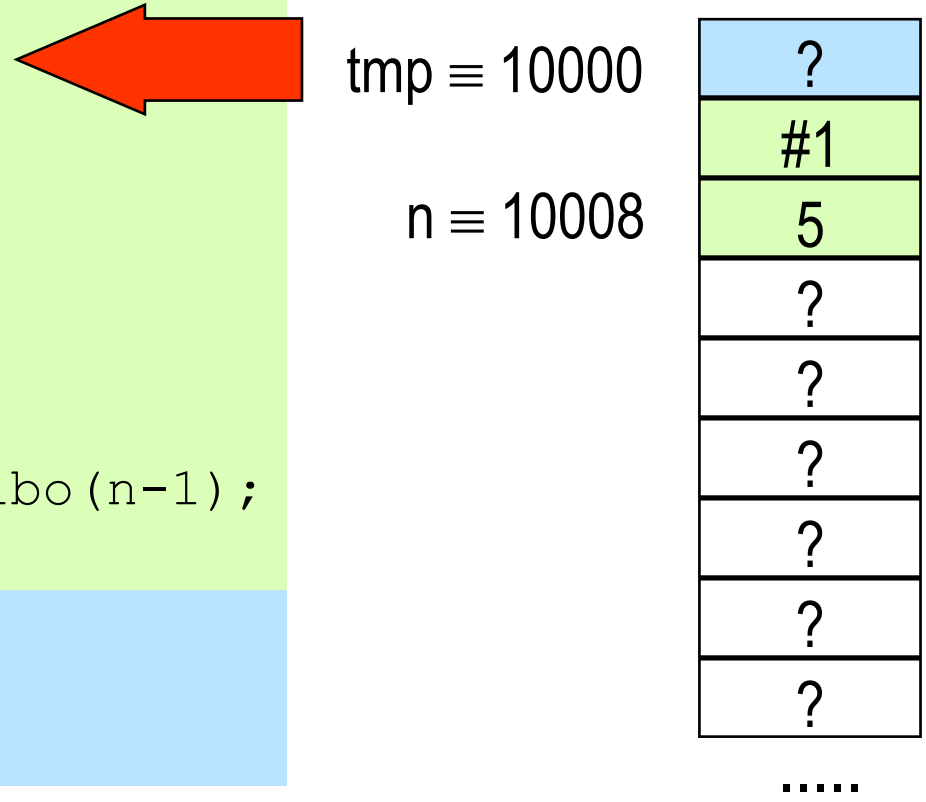
#1



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

#1



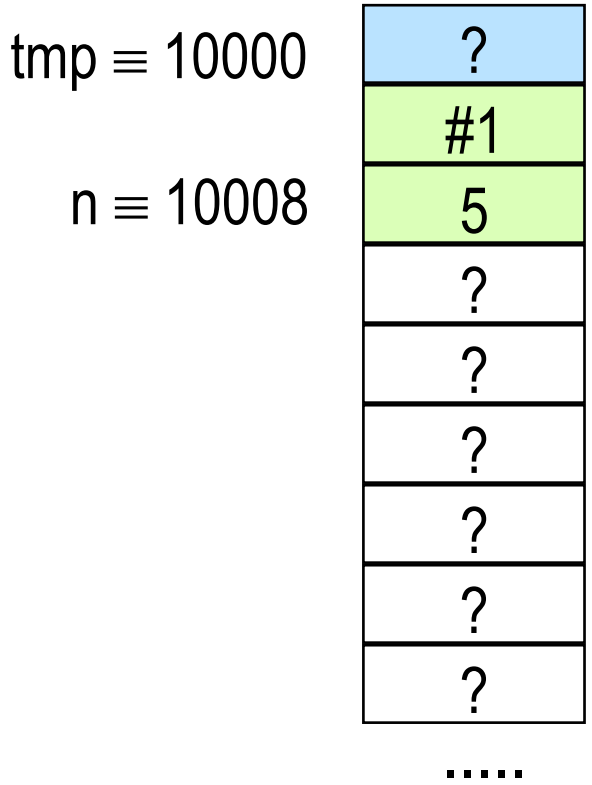
universität wien

Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
```

```
int main () {
  cout << fibo(5) << '\n';
}
```

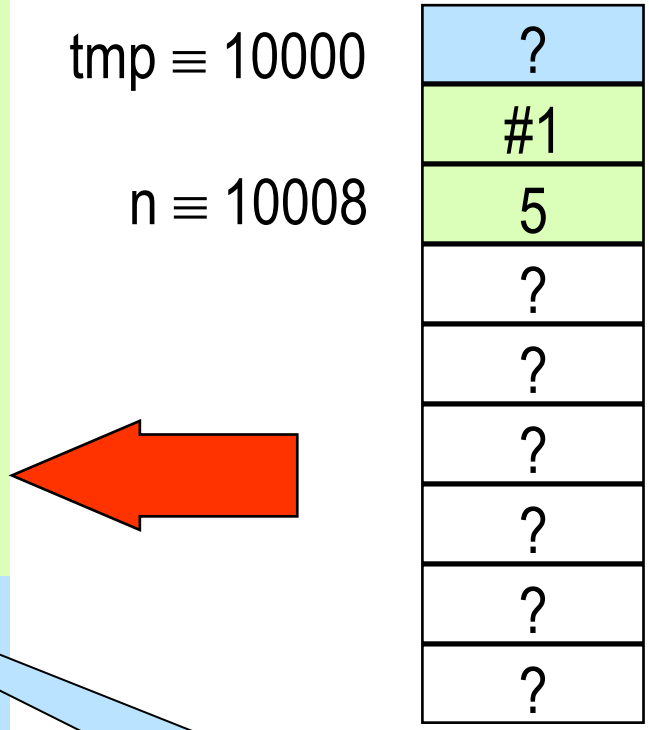
#1



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

#1



Annahme: linker Teilausdruck wird zuerst bewertet

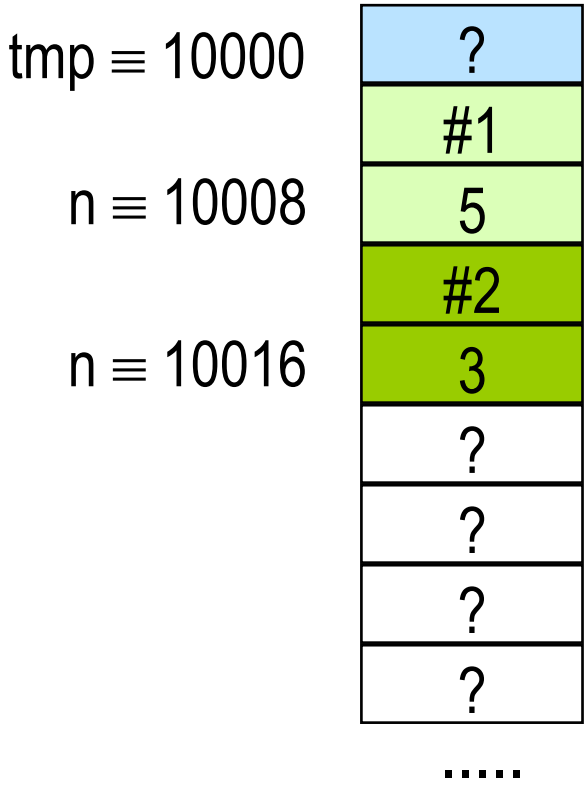
Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
```

#2

```
int main () {
  cout << fibo(5) << '\n';
}
```

#1



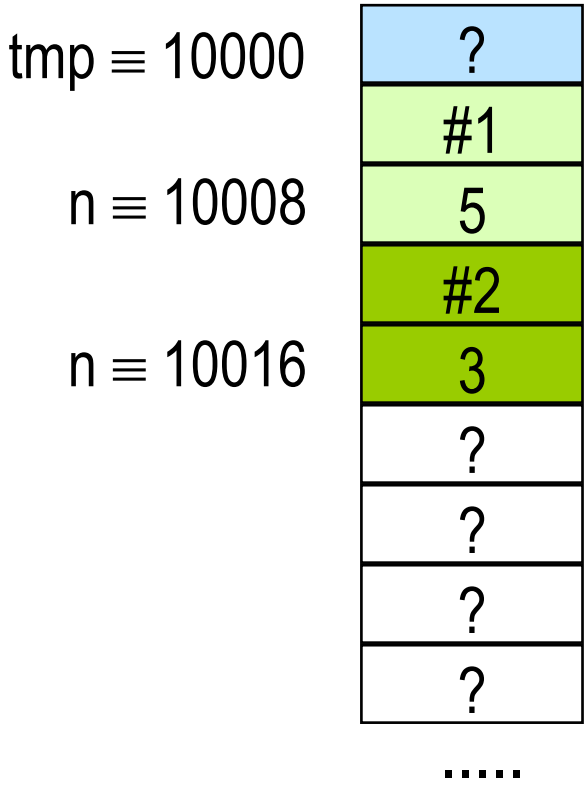
Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
```

#2

```
int main () {
  cout << fibo(5) << '\n';
}
```

#1



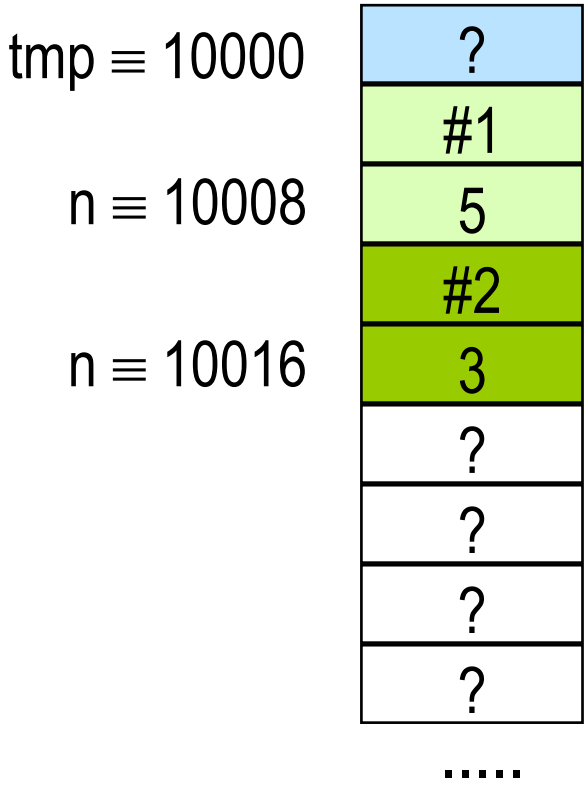
Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
```

#2

```
int main () {
  cout << fibo(5) << '\n';
}
```

#1

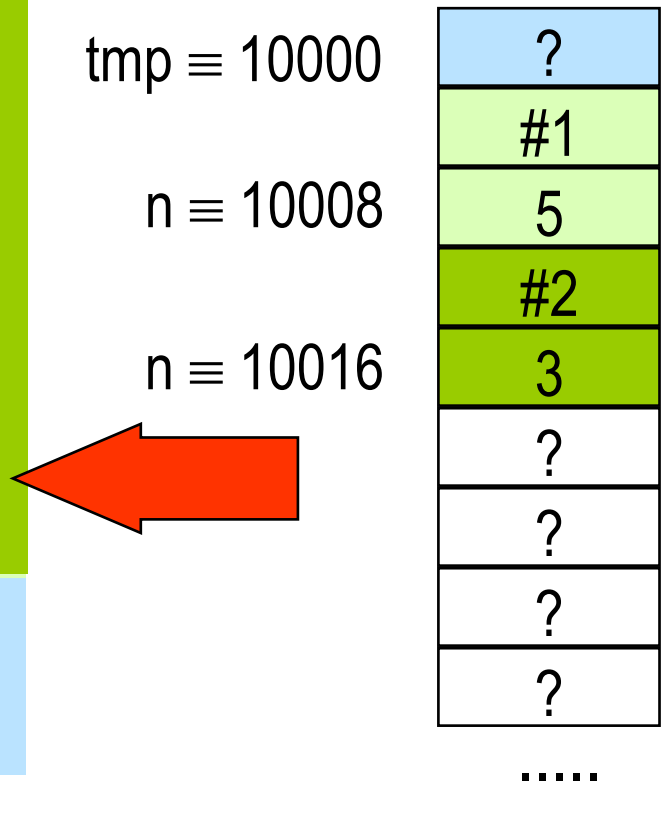


Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

#2

#1



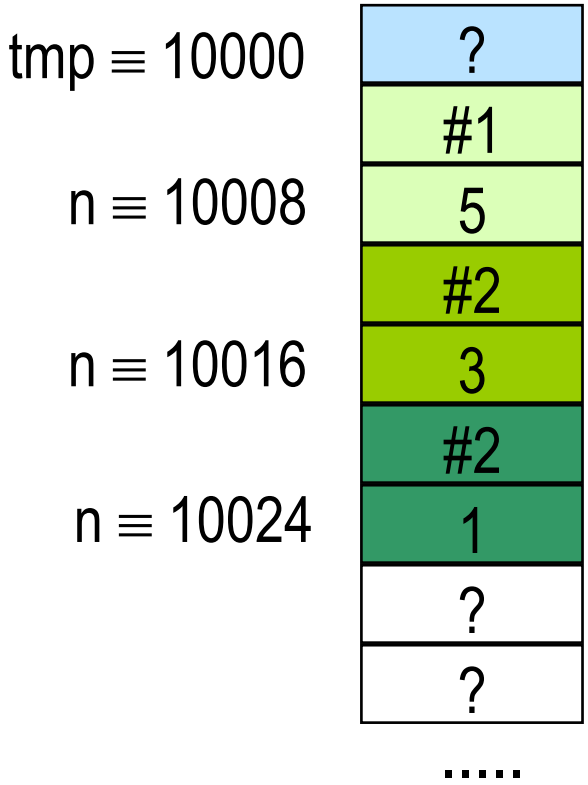
Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
```

#2

```
int main () {
  cout << fibo(5) << '\n';
}
```

#1



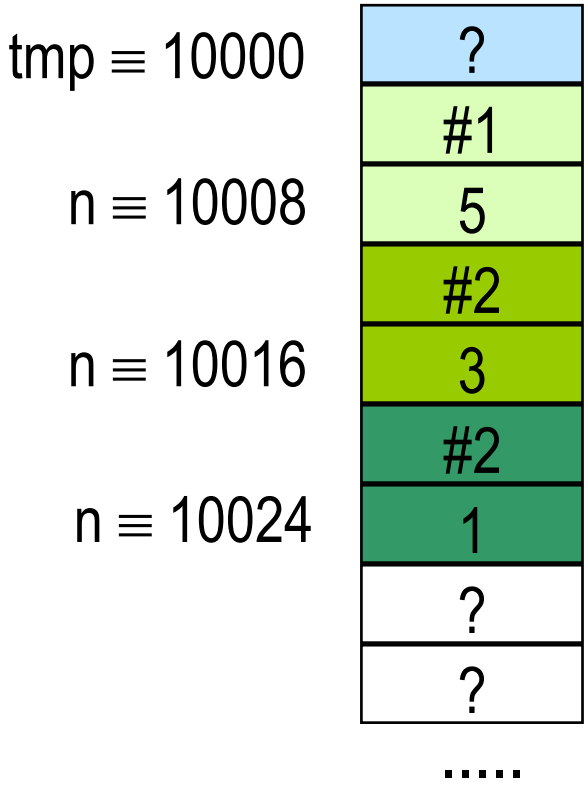
Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
```

#2

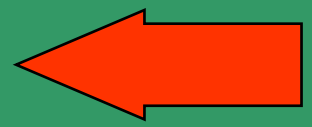
```
int main () {
  cout << fibo(5) << '\n';
}
```

#1



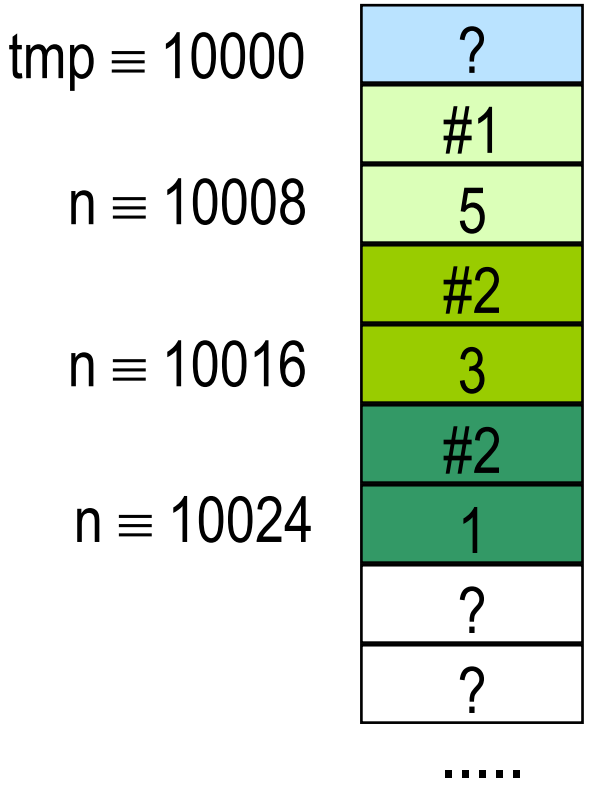
Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```



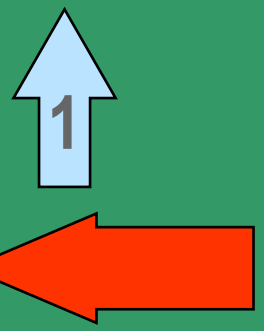
#2

#1



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```



#2

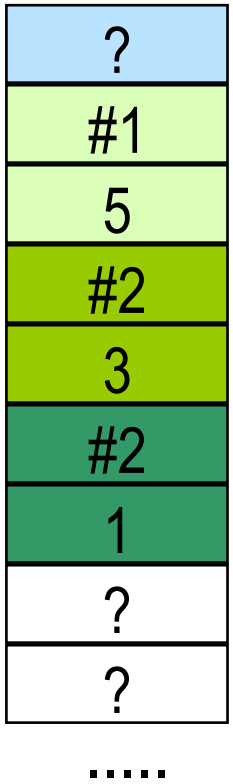
#1

tmp ≡ 10000

n ≡ 10008

n ≡ 10016

n ≡ 10024



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
      return 1;
    else
      return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

≡ 1

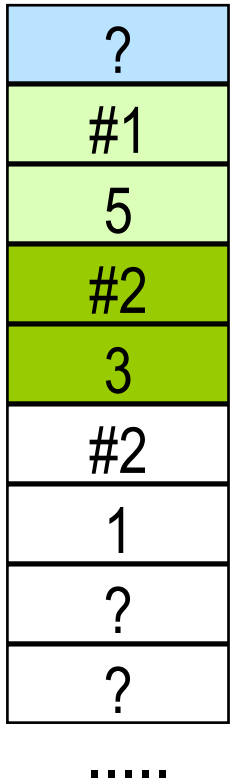
#2

#1

tmp ≡ 10000

n ≡ 10008

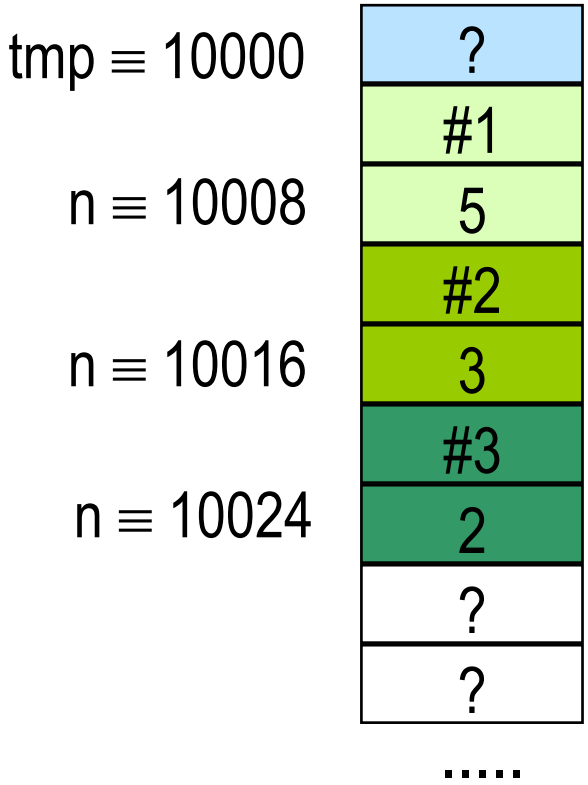
n ≡ 10016



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

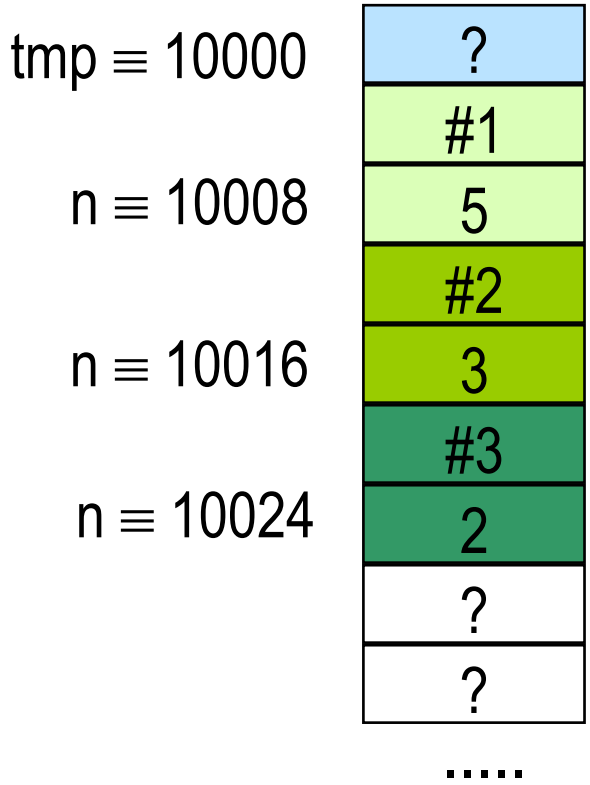
#1 #2 #3



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

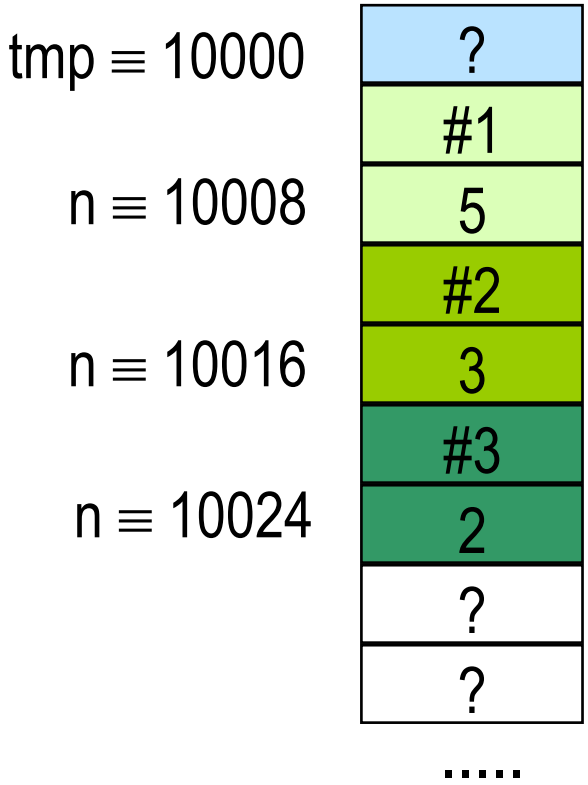
#1 #2 #3



Direkte Rekursion: Fibonacci Zahlen (1)

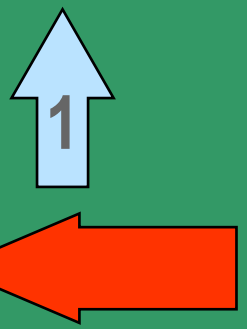
```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

#1 #2 #3

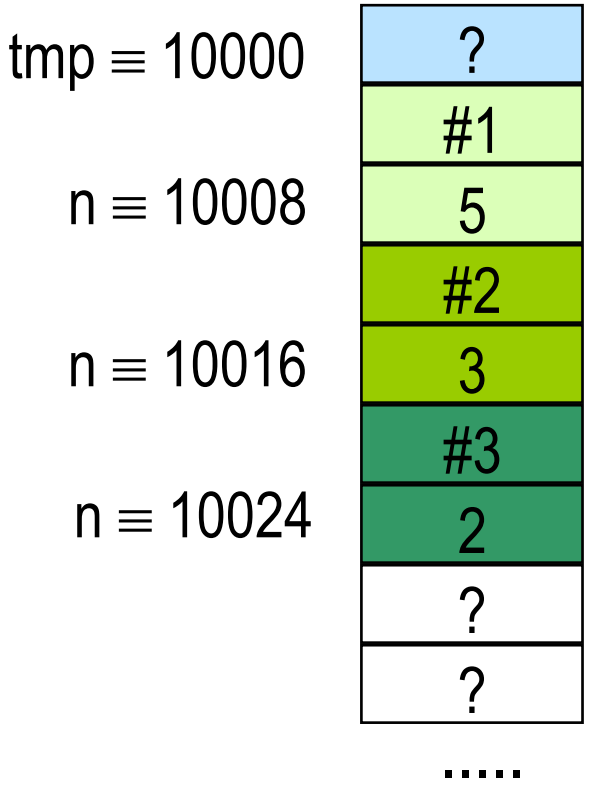


Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

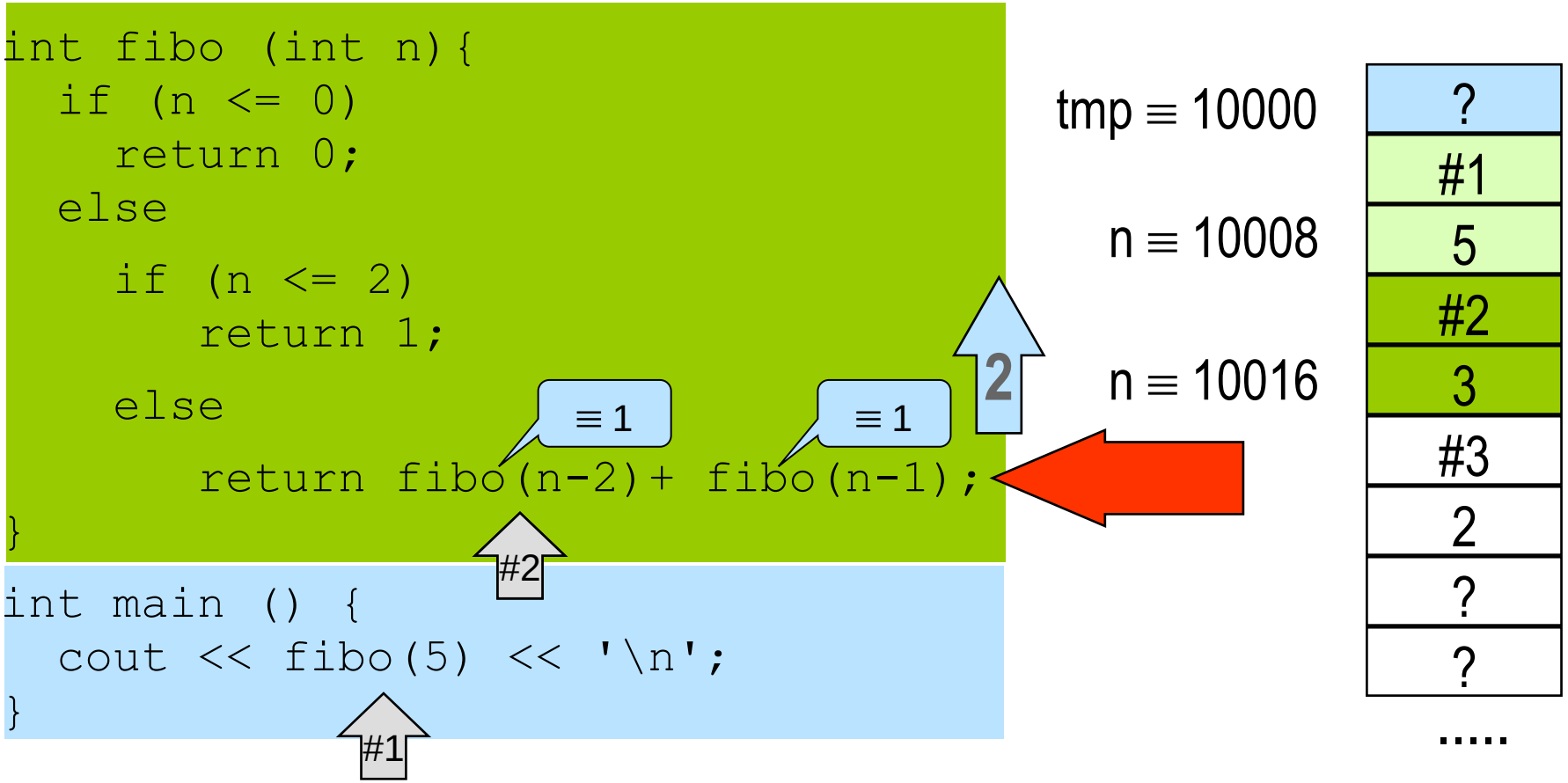


#2 #3 #1



Direkte Rekursion: Fibonacci Zahlen (1)

```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
      return 1;
    else
      return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```



Direkte Rekursion: Fibonacci Zahlen (1)

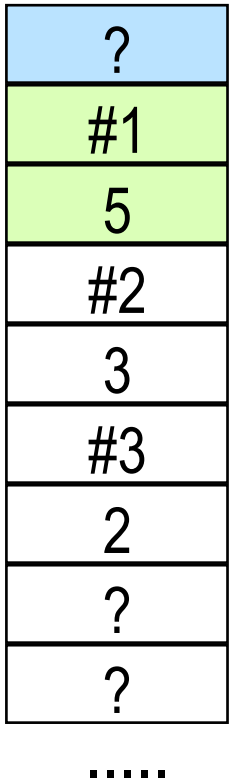
```
int fibo (int n){
  if (n <= 0)
    return 0;
  else
    if (n <= 2)
       return 1;
    else
       return fibo(n-2)+ fibo(n-1);
}
int main () {
  cout << fibo(5) << '\n';
}
```

≡ 2

#1

tmp ≡ 10000

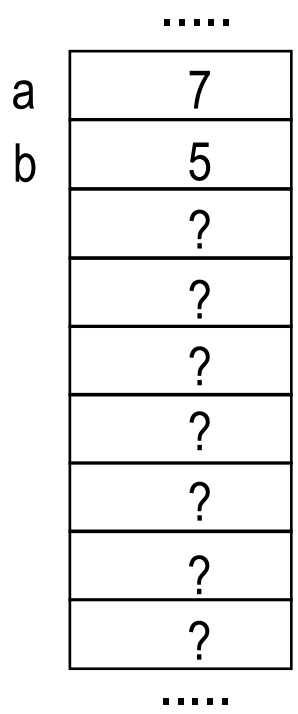
n ≡ 10008



Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a

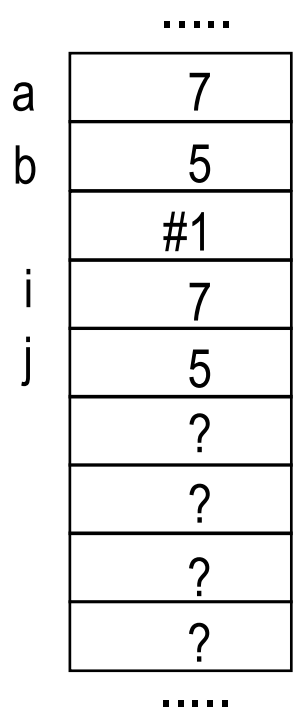


universität wien

Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

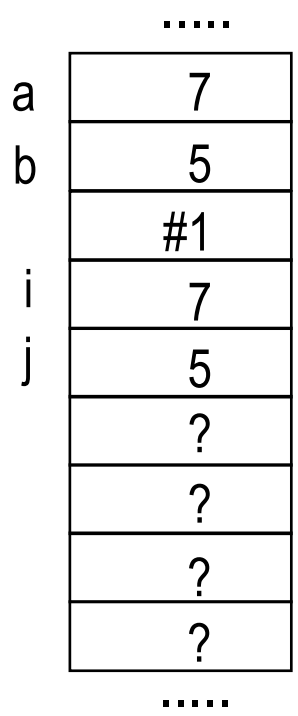
Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a

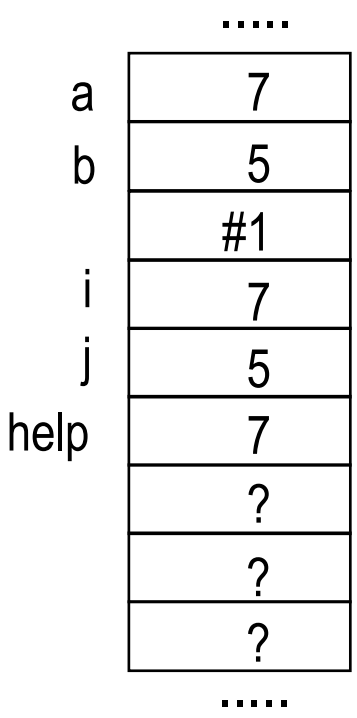


universität wien

Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

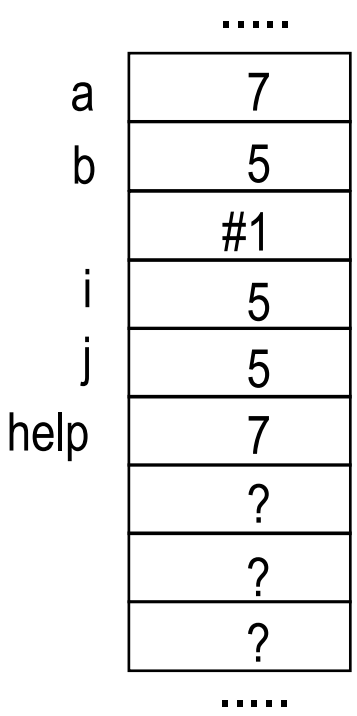
Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

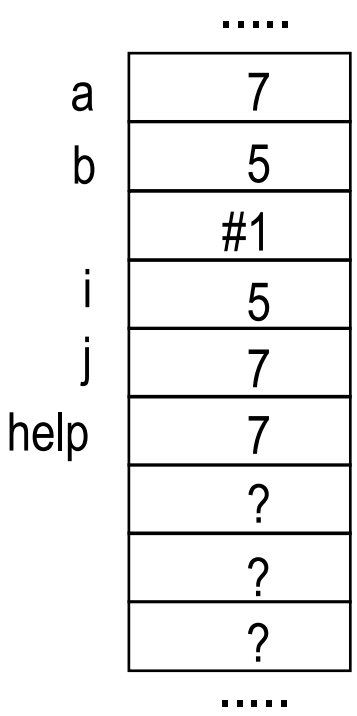
Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



Wertparameter – call by value

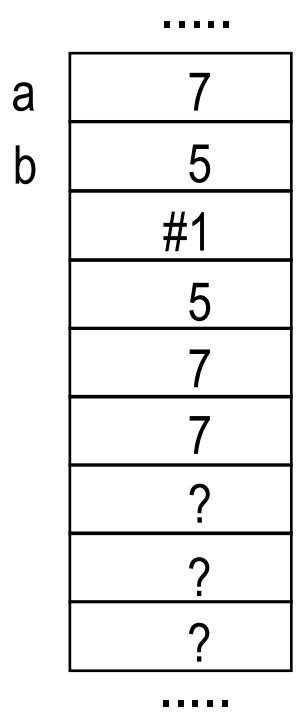
```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a

Wertparameter – call by value

```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

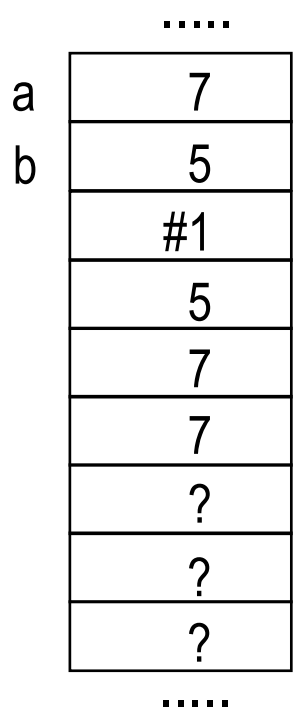
Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



Wertparameter – call by value

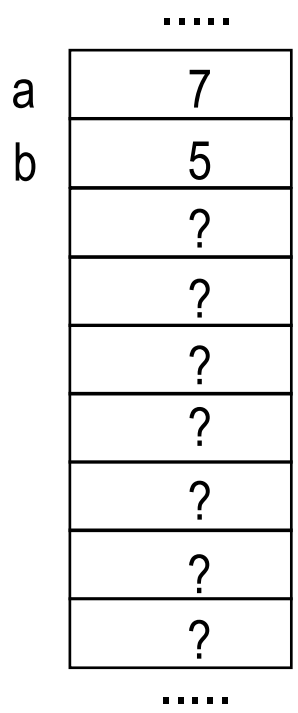
```
void sort(int i, int j) { //soll die Parameterwerte sortieren
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```

Scheitert, wenn für b ein kleinerer Wert eingegeben wurde, als für a



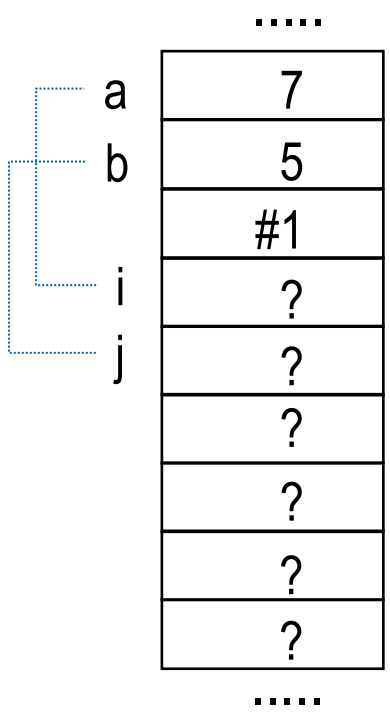
Referenzparameter – call by reference

```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



Referenzparameter – call by reference

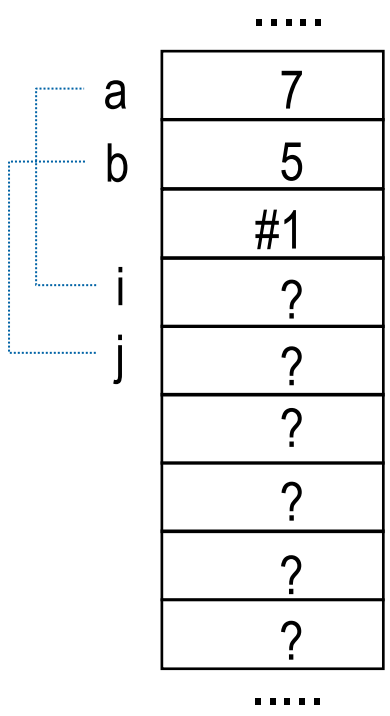
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

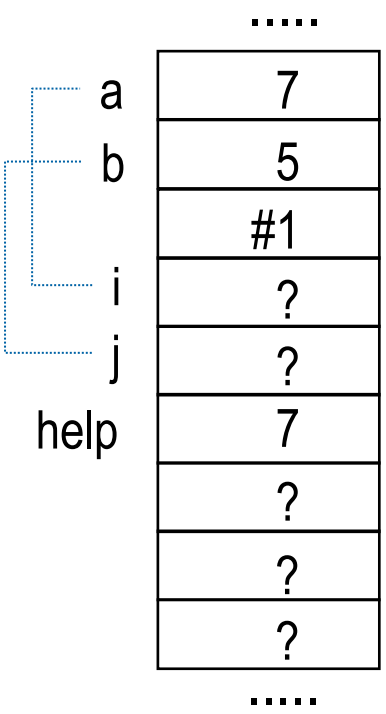
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

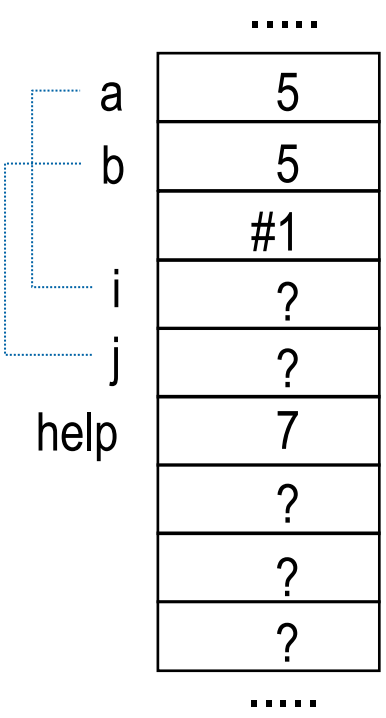
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

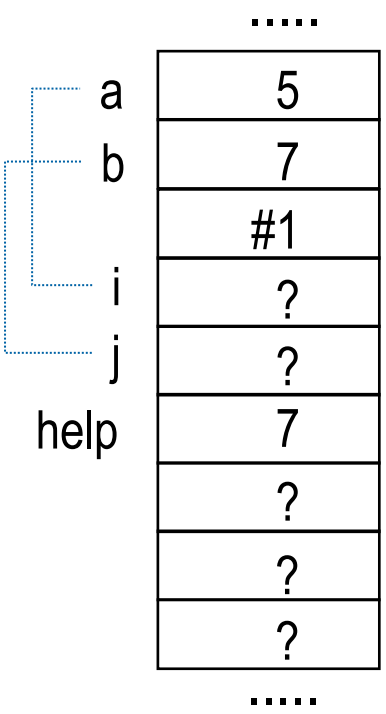
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

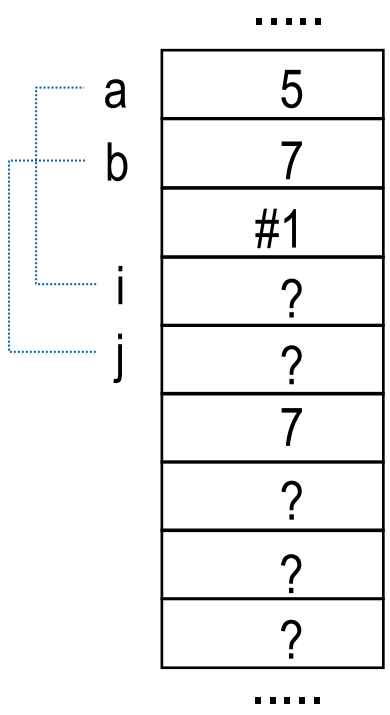
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

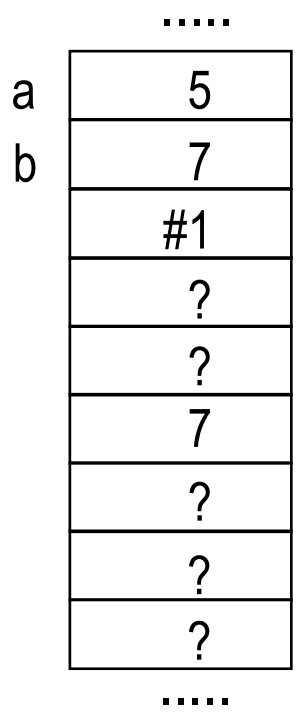
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

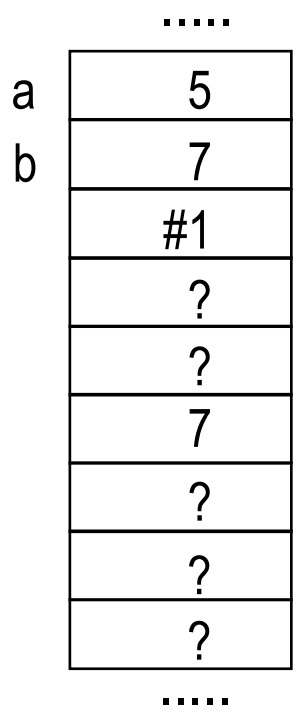
```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Referenzparameter – call by reference

```
void sort(int& i, int& j) {
  if (i>j) {
    int help {i};
    i = j;
    j = help;
  }
}
int main() {
  int a,b;
  cin >> a >> b;
  sort(a,b);
  assert(a<=b);
  return 0;
}
```



universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

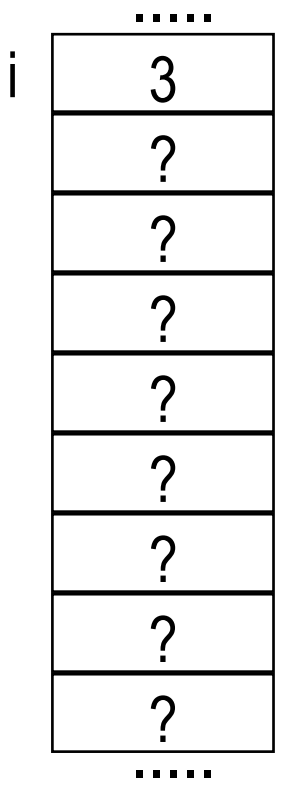
Compiler Warnung:
converting to <int>
from <double>

universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

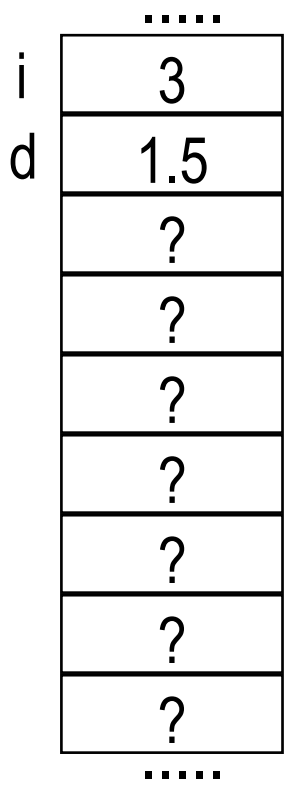


universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

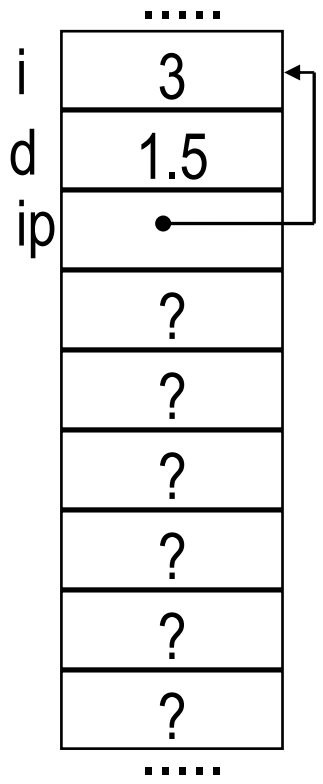


universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

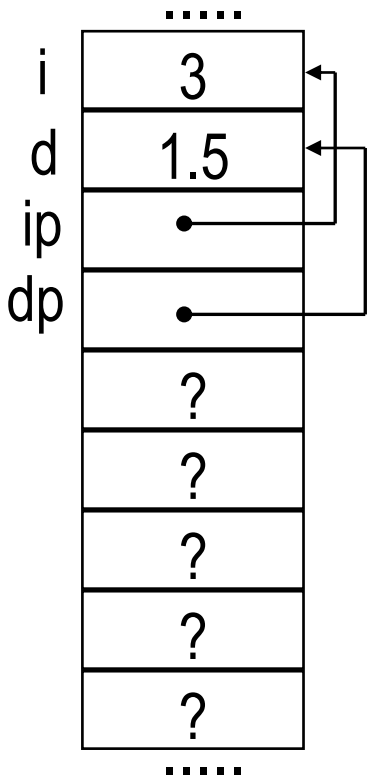


universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

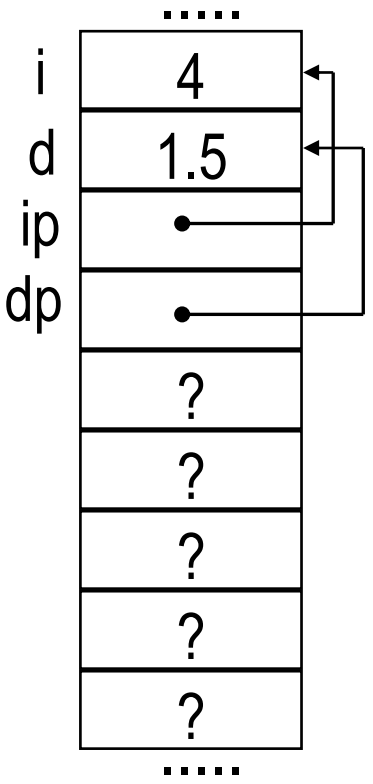


universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

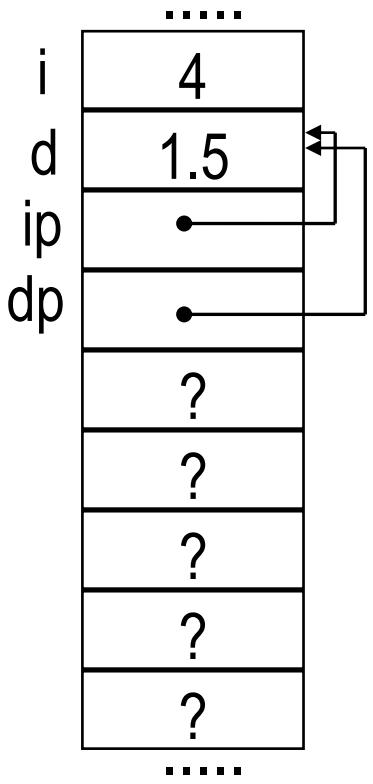


universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```



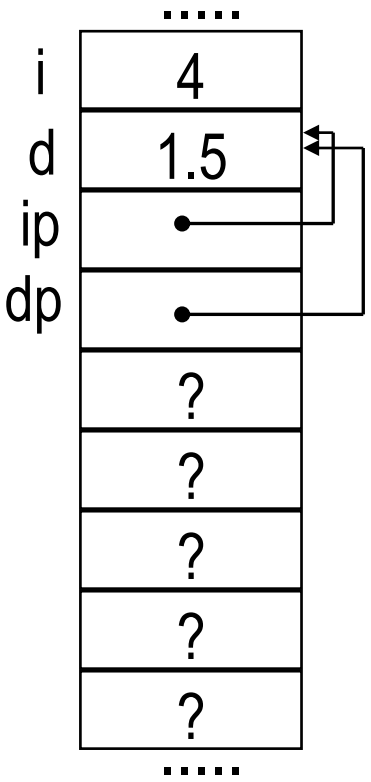
universität wien

Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

Ausgabe: **0**

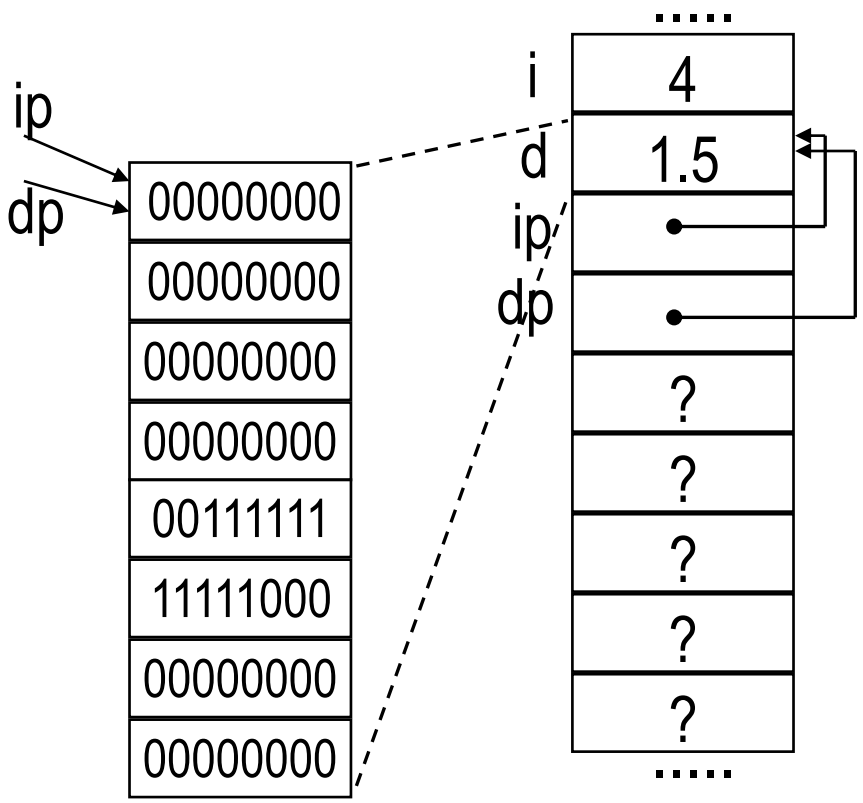


Zeiger (5)

```
int i {3};
double d {1.5};
int *ip {&i};
double *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

Ausgabe: **0**

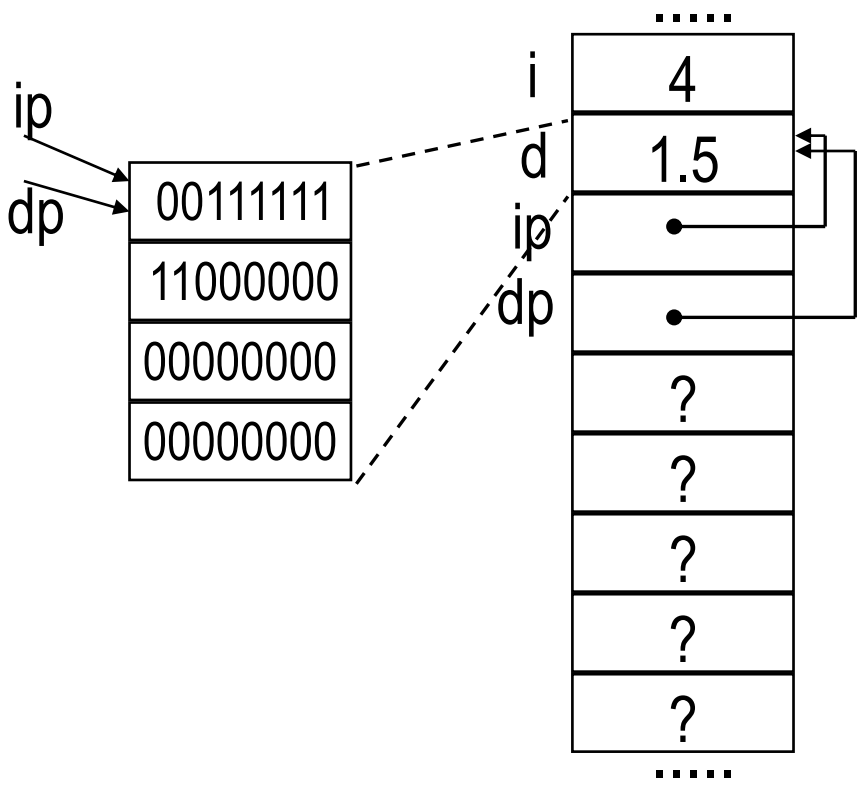


Zeiger (5)

```
int i {3};
float d {1.5};
int *ip {&i};
float *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

Verwendung von float, damit beide Datentypen gleich lang sind.



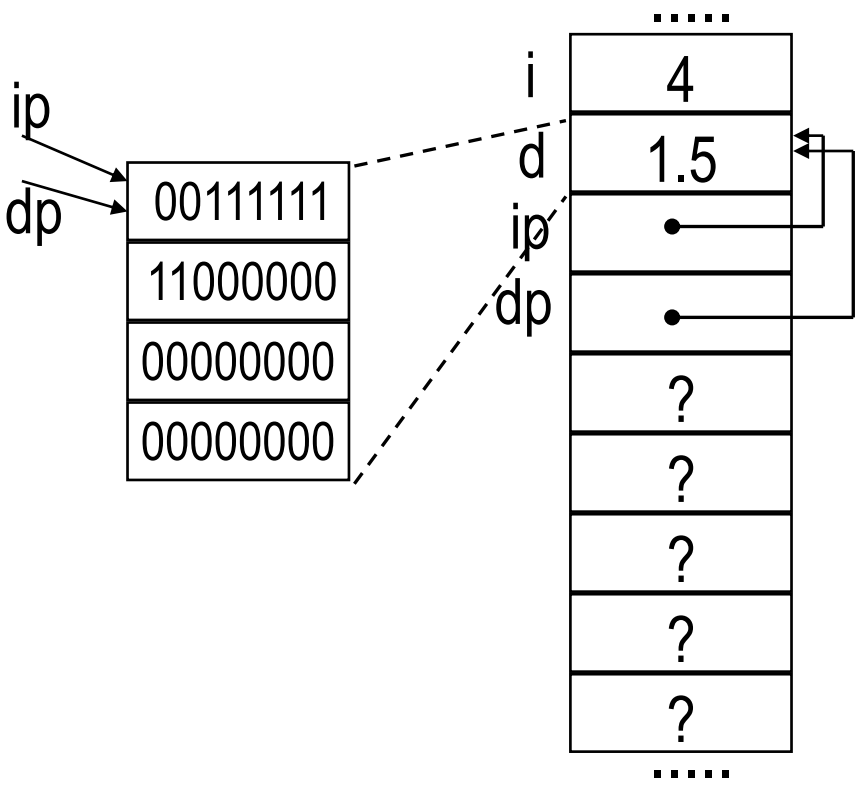
Zeiger (5)

```
int i {3};
float d {1.5};
int *ip {&
float *dp {&d};

*ip = *ip + *dp;
ip = (int *)dp;
cout << *ip;
```

Verwendung von float, damit beide Datentypen gleich lang sind. Hilft auch nicht!

Ausgabe:
1069547520



universität wien

```
int main() {
  cout << "Zeigerverbiegungen\n\n";
  int *ip;
  { // warum wohl diese Klammern ??
    cout << "Der Tragoedie erster Teil: Datentyp int\n";
    int v {4711};
    int *pv {&v};
    int **ppv {&pv};
    cout << "Die Werte sind: " << v << ", " << (void *)pv << " und " << (void *)ppv << '\n';
    cout << "v = " << v << " = " << *pv << " = " << **ppv << '\n';
    int f[10]{0,1,2,3,4,5,6,7,8,9};
    for (int i{0}; i < 10; i=i+1)
      cout << "Index: " << i << " Wert: " << f[i] << " Adresse: " << (void *)(f + i) << '\n';
    pv = f + 9; // Aequivalent zu &f[9]
    cout << "Letzter Wert indirekt: " << *pv << '\n';
    ip = new int; // warum nicht ip = &v?
    *ip = v;
  }
```

```
Zeigerverbiegungen

Der Tragoedie erster Teil: Datentyp int
Die Werte sind: 4711, 0x7fff232dbdbc und
0x7fff232dbda8
v = 4711 = 4711 = 4711
Index: 0 Wert: 0 Adresse: 0x7fff232dbd80
Index: 1 Wert: 1 Adresse: 0x7fff232dbd84
Index: 2 Wert: 2 Adresse: 0x7fff232dbd88
Index: 3 Wert: 3 Adresse: 0x7fff232dbd8c
Index: 4 Wert: 4 Adresse: 0x7fff232dbd90
Index: 5 Wert: 5 Adresse: 0x7fff232dbd94
Index: 6 Wert: 6 Adresse: 0x7fff232dbd98
Index: 7 Wert: 7 Adresse: 0x7fff232dbd9c
Index: 8 Wert: 8 Adresse: 0x7fff232dbda0
Index: 9 Wert: 9 Adresse: 0x7fff232dbda4
Letzter Wert indirekt: 9
```

universität wien

```
{
   cout << "\nFunktioniert aber auch mit double\n";
   double v {47.11};
   double *pv {&v};
   double **ppv {&pv};
   cout << "Die Werte sind: " << v << ", " << (void *)pv << " und " << (void *)ppv << '\n';
   cout << "v = " << v << " = " << *pv << " = " << **ppv << '\n';
   double f[10]{0,0.5,1,1.5,2,2.5,3,3.5,4,4.5};
   for (int i{0}; i < 10; i=i+1)
     cout << "Index: " << i << " Wert: " << f[i] << " Adresse: " << (void *)(f + i) << '\n';
   pv = f + 9; // Aequivalent zu &f[9]
   cout << "Letzter Wert indirekt: " << *pv << '\n';
   cout << "Dynamisches Element: " << *ip << '\n';
}
```

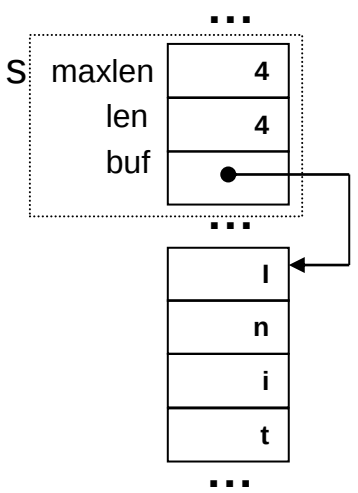
```
Funktioniert aber auch mit double
Die Werte sind: 47.11, 0x7fff232dbdb0 und 0x7fff232dbda8
v = 47.11 = 47.11 = 47.11
Index: 0 Wert: 0 Adresse: 0x7fff232dbd30
Index: 1 Wert: 0.5 Adresse: 0x7fff232dbd38
Index: 2 Wert: 1 Adresse: 0x7fff232dbd40
Index: 3 Wert: 1.5 Adresse: 0x7fff232dbd48
Index: 4 Wert: 2 Adresse: 0x7fff232dbd50
Index: 5 Wert: 2.5 Adresse: 0x7fff232dbd58
Index: 6 Wert: 3 Adresse: 0x7fff232dbd60
Index: 7 Wert: 3.5 Adresse: 0x7fff232dbd68
Index: 8 Wert: 4 Adresse: 0x7fff232dbd70
Index: 9 Wert: 4.5 Adresse: 0x7fff232dbd78
Letzter Wert indirekt: 4.5
Dynamisches Element: 4711
```

```
{
   cout << "\nUnd erst recht mit char\n";
   char v {'x'};
   char *pv {&v};
   char **ppv {&pv};
   cout << "Die Werte sind: " << v << ", " << (void *)pv << " und " << (void *)ppv << '\n';
   cout << "v = " << v << " = " << *pv << " = " << **ppv << '\n';
   char f[10]{'z','a','y','b','x','c','w','d','v','e'};
   for (int i{0}; i < 10; i=i+1)
     cout << "Index: " << i << " Wert: " << f[i] << " Adresse: " << (void *)(f + i) << '\n';
   pv = f + 9; // Aequivalent zu &f[9]
   cout << "Letzter Wert indirekt: " << *pv << '\n';
   delete ip;
 }
 cout << "\n\n";
 return(0);
}
```

```
Und erst recht mit char
Die Werte sind: x, 0x7fff232dbdbc und 0x7fff232dbda8
v = x = x = x
Index: 0 Wert: z Adresse: 0x7fff232dbdc0
Index: 1 Wert: a Adresse: 0x7fff232dbdc1
Index: 2 Wert: y Adresse: 0x7fff232dbdc2
Index: 3 Wert: b Adresse: 0x7fff232dbdc3
Index: 4 Wert: x Adresse: 0x7fff232dbdc4
Index: 5 Wert: c Adresse: 0x7fff232dbdc5
Index: 6 Wert: w Adresse: 0x7fff232dbdc6
Index: 7 Wert: d Adresse: 0x7fff232dbdc7
Index: 8 Wert: v Adresse: 0x7fff232dbdc8
Index: 9 Wert: e Adresse: 0x7fff232dbdc9
Letzter Wert indirekt: e
```

Weitere Probleme mit Klasse String

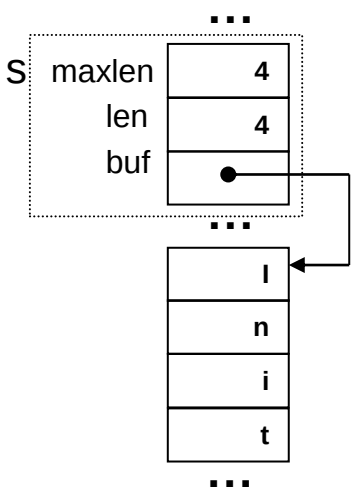
```
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

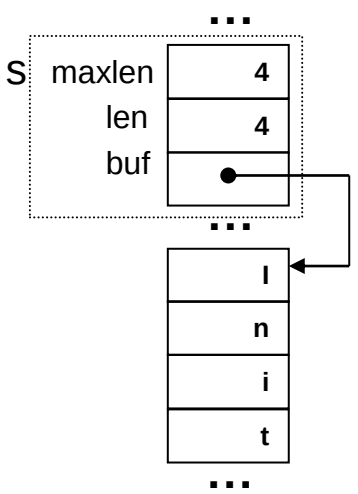
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

main() {
  String s {"Init"};
  (s-"ni").print();
}
```



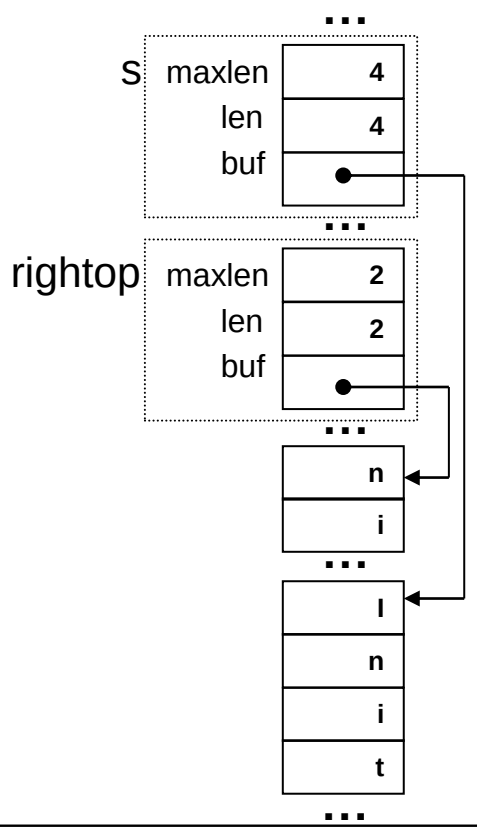
Konstruktoraufruf

Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

main() {
  String s {"Init"};
  (s-"ni").print();
}
```

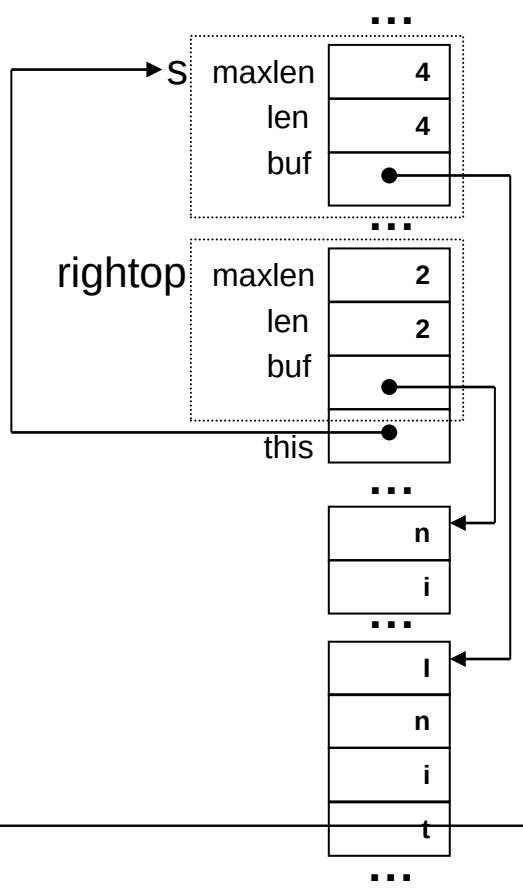
Erstes Argument via **this**



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

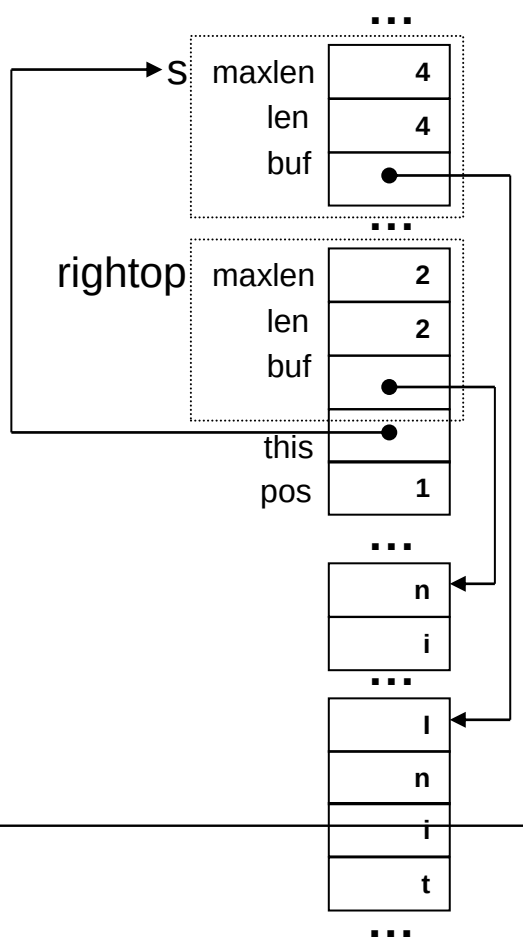
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

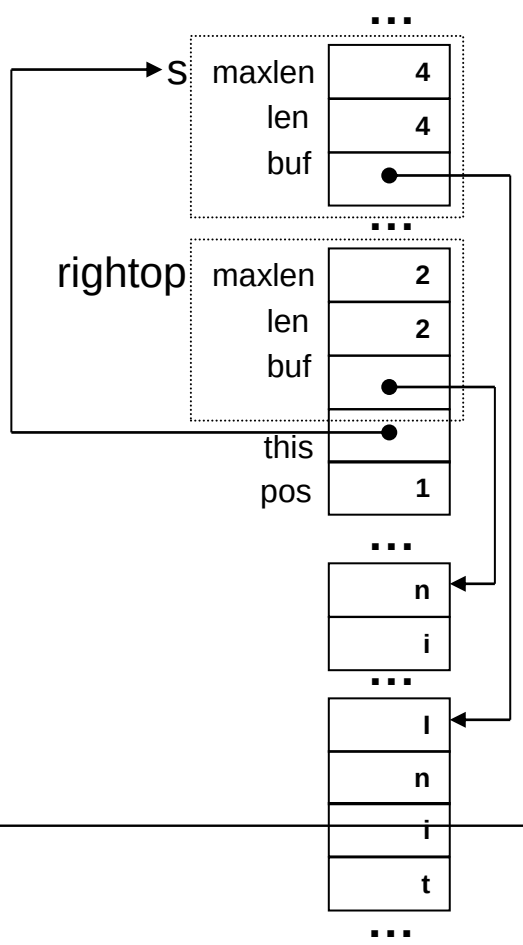
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

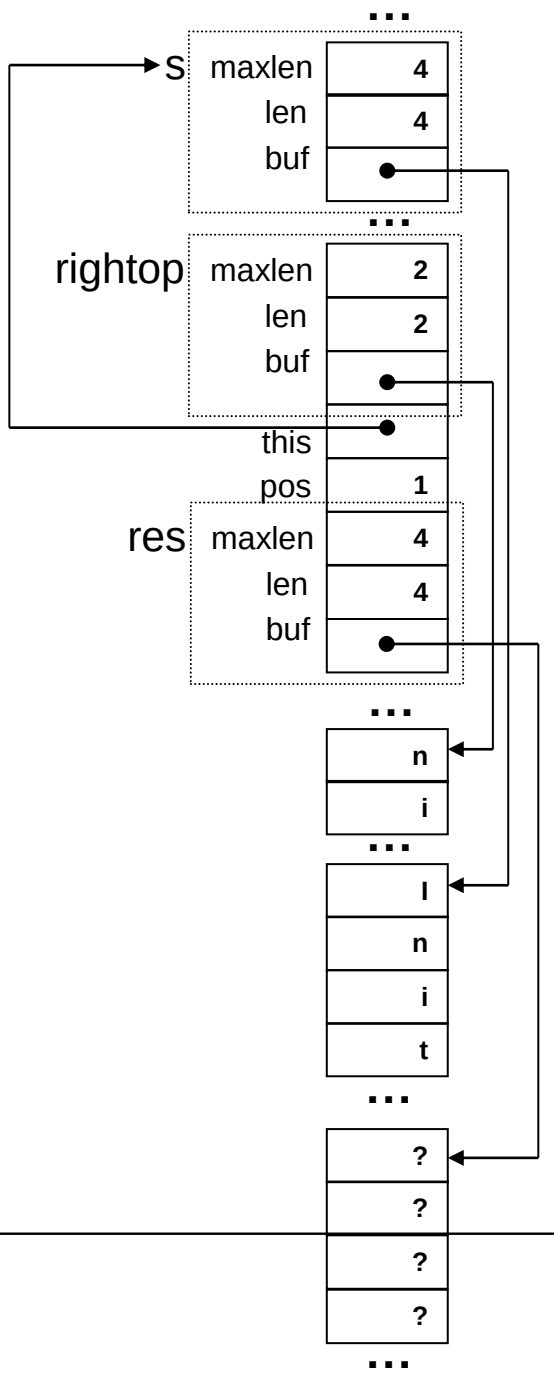
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

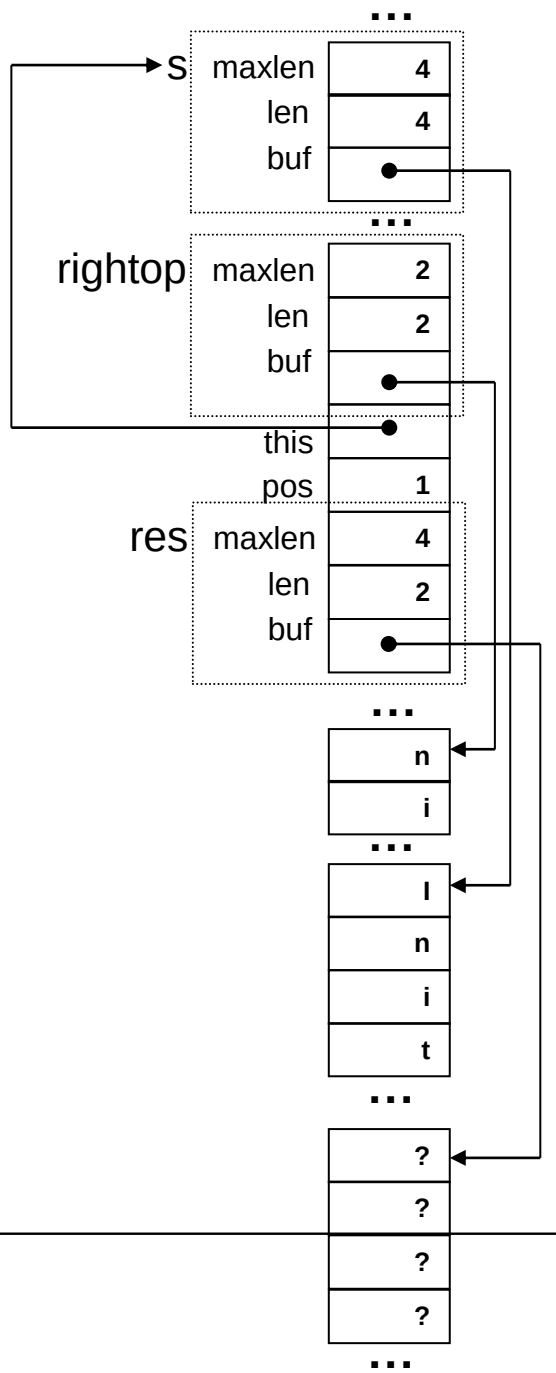
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

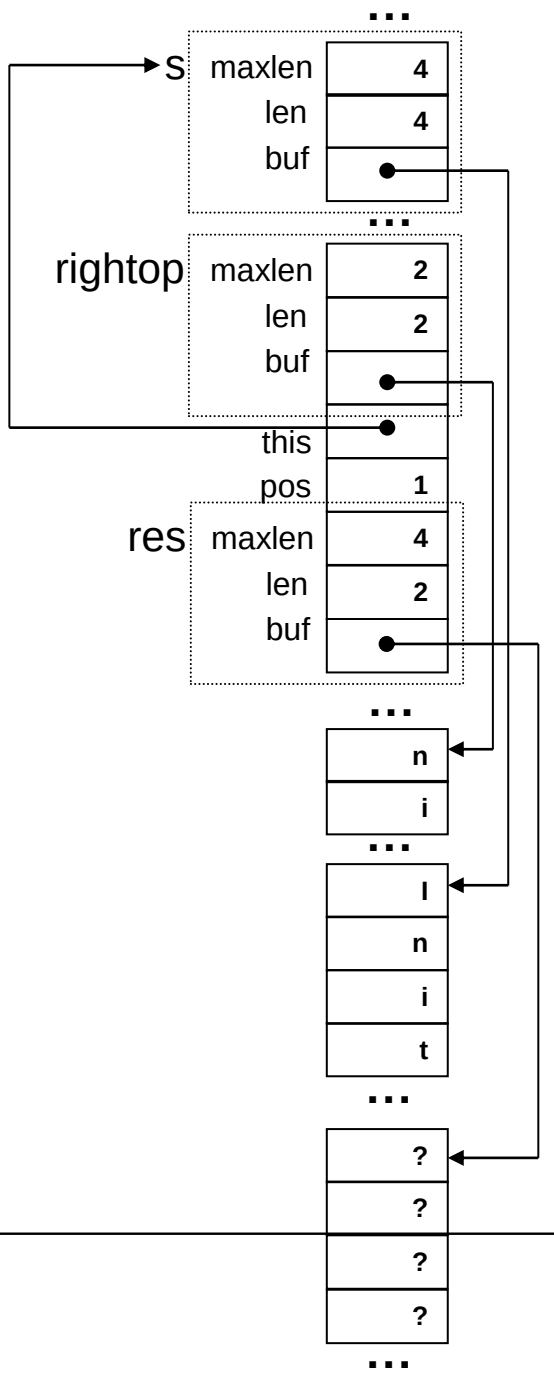
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

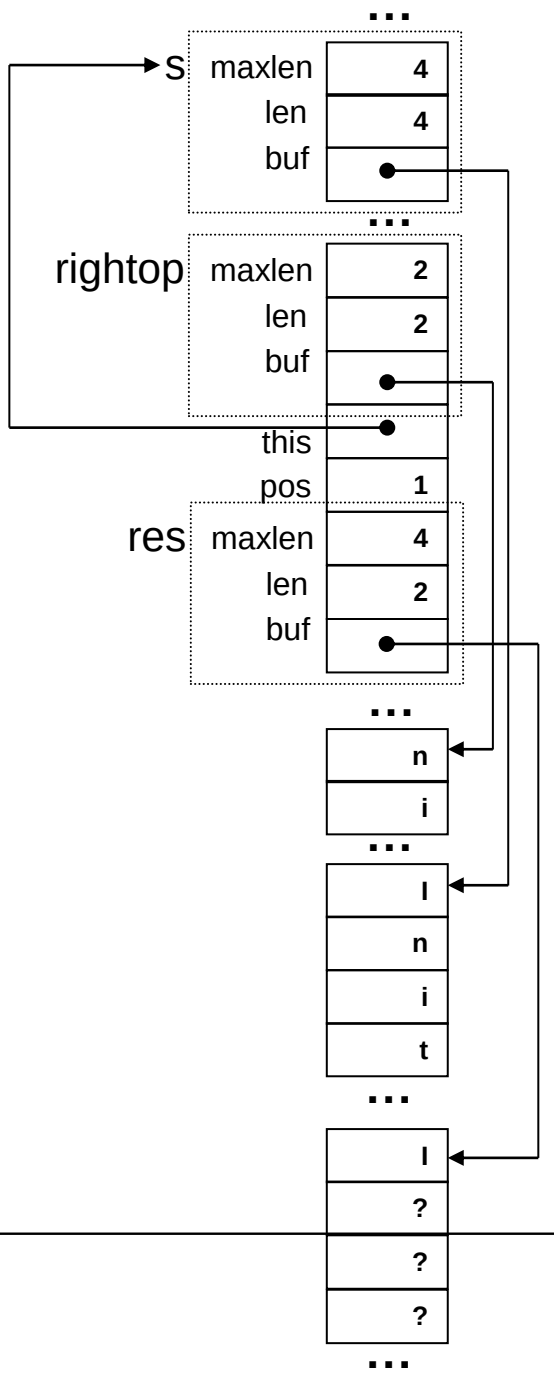
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

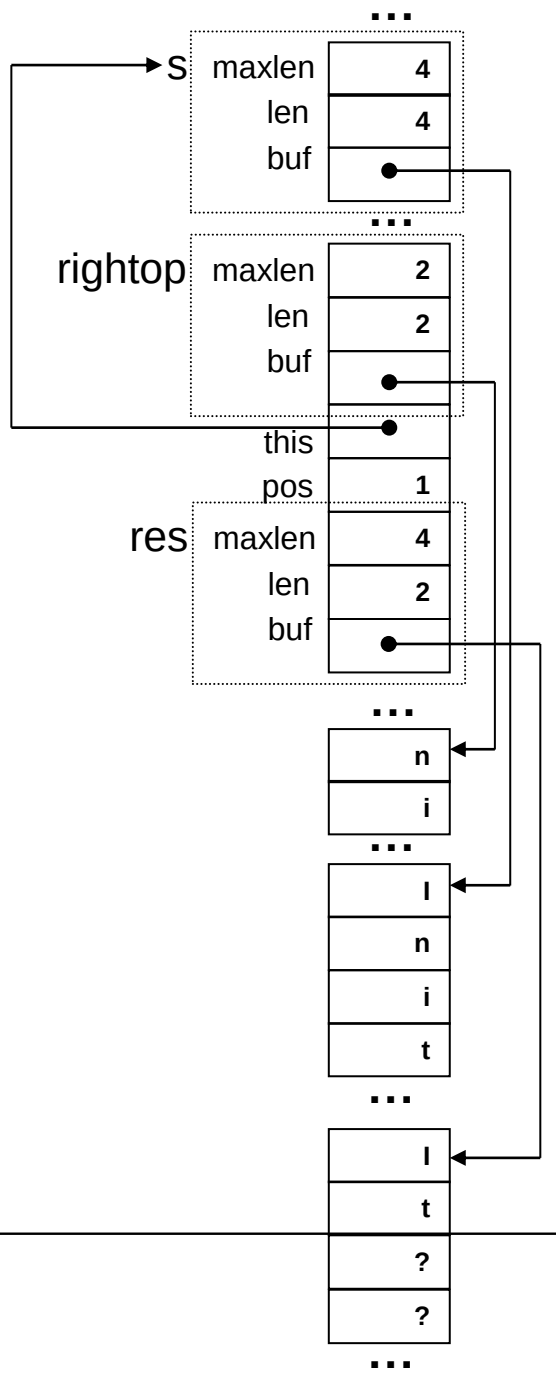
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

main() {
  String s {"Init"};
  (s-"ni").print();
}
```

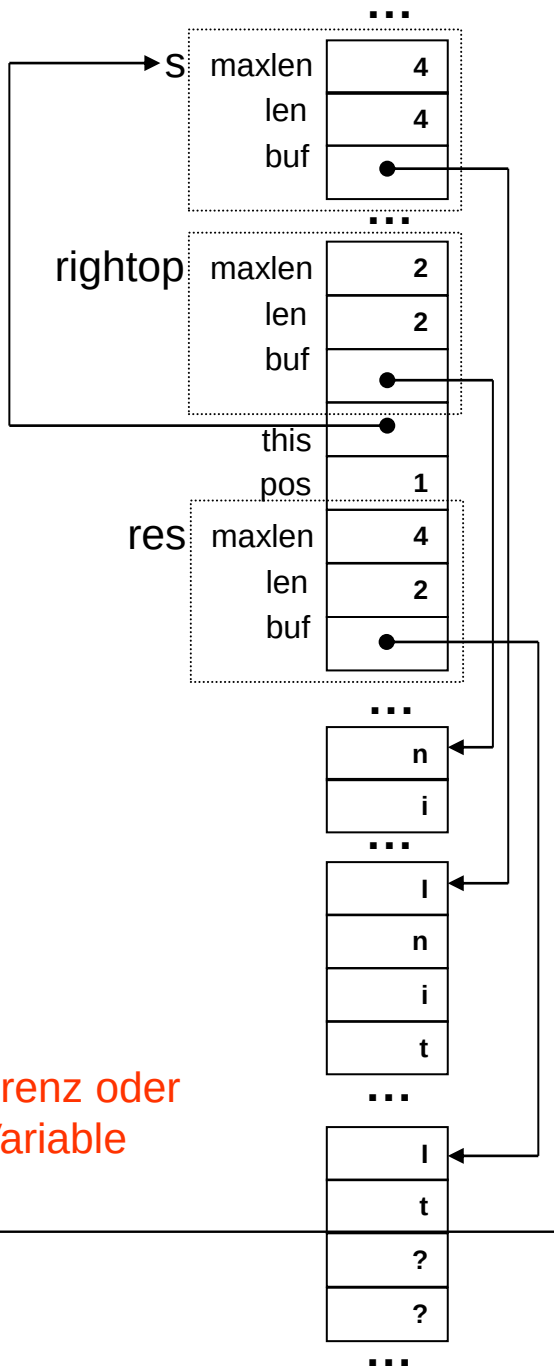


Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

main() {
  String s {"Init"};
  (s-"ni").print();
}
```

Es muss eine (temporäre) Kopie erstellt werden (Warum?).



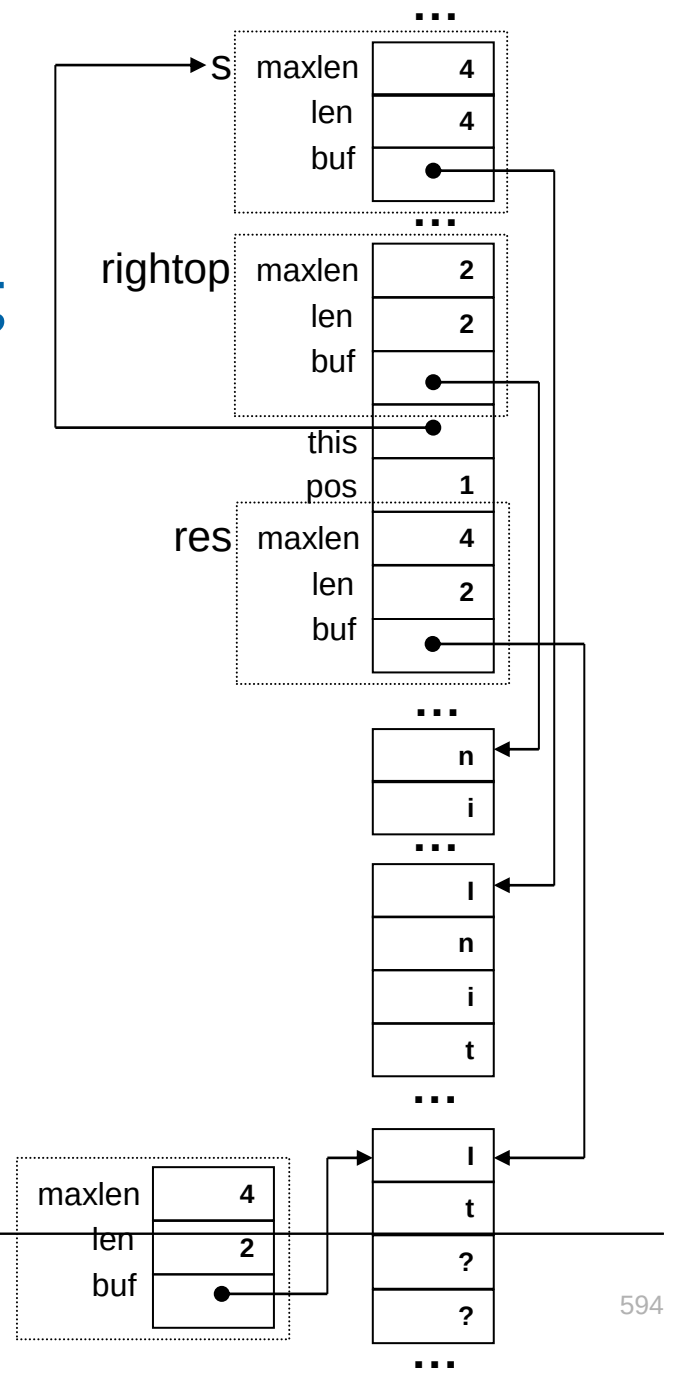
Merkregel: Niemals eine Referenz oder einen Zeiger auf eine lokale Variable retournieren!

Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

main() {
  String s {"Init"};
  (s-"ni").print();
}
```

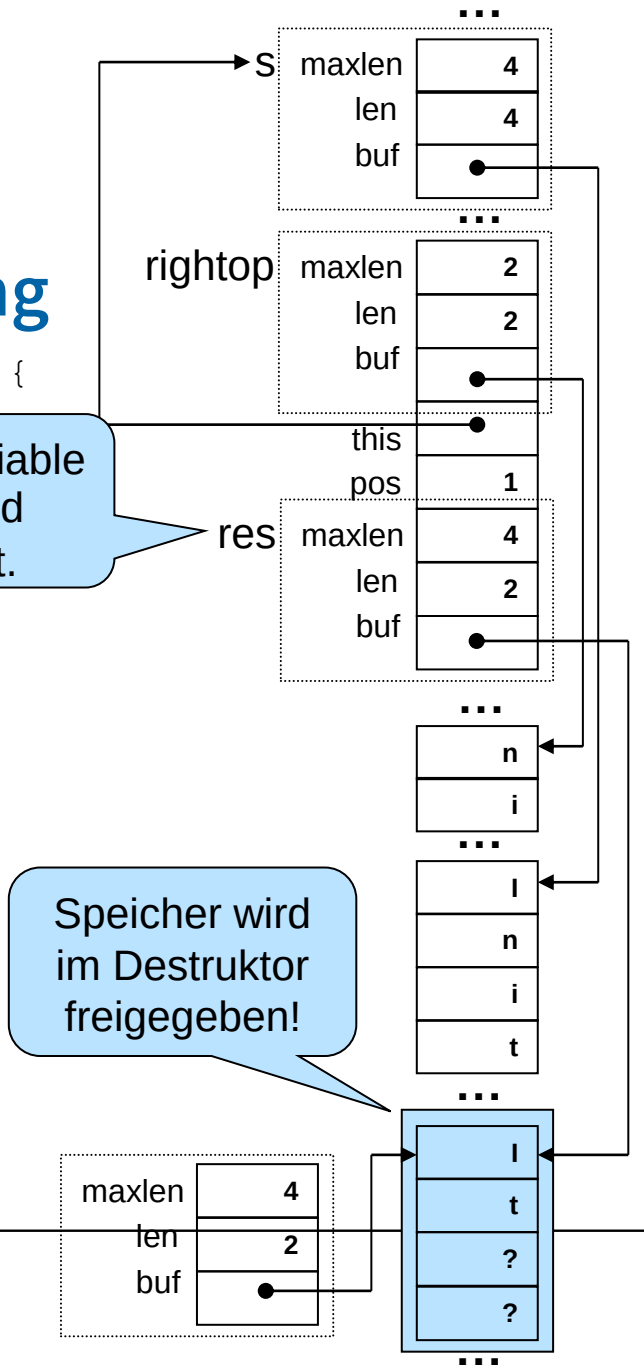
Komponentenweise Kopie



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

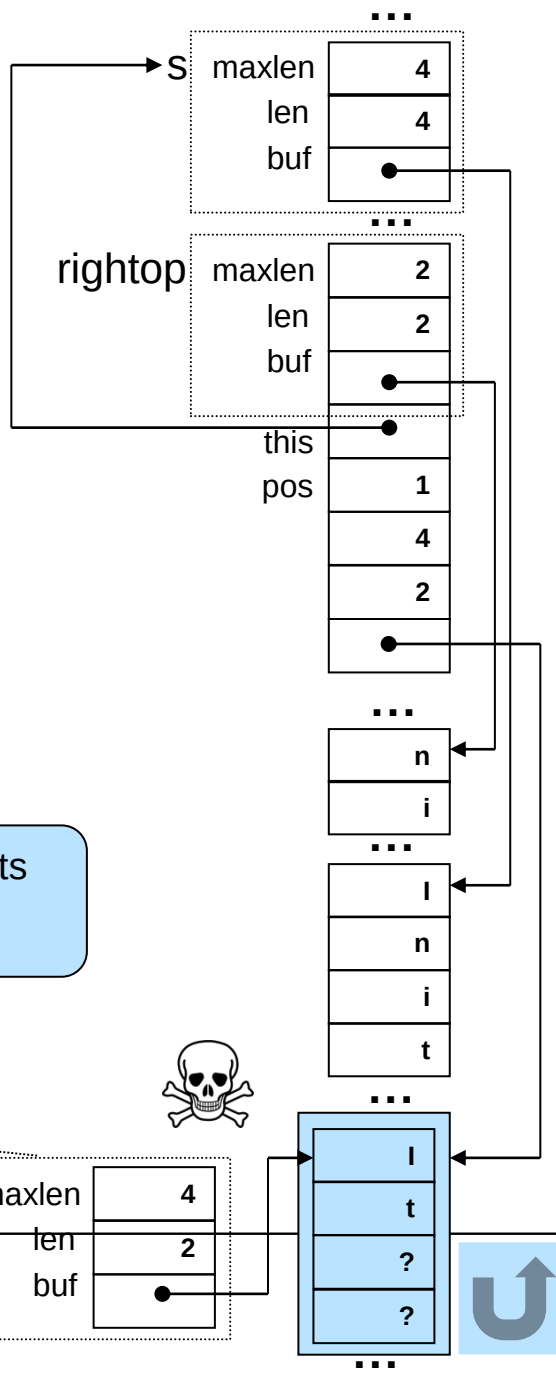
main() {
  String s {"Init"};
  (s-"ni").print();
}
```



Weitere Probleme mit Klasse String

```
String String::operator- (const String& rightop) {
  int pos {this->find(rightop)};
  if (pos>=0) {
    String res{maxlen};
    res.len=len-rightop.len;
    assert(res.len>=0);
    for(int i=0; i<pos; ++i) res.buf[i] = buf[i];
    for(int i=pos+rightop.len; i<len; ++i)
      res.buf[i-rightop.len] = buf[i];
    return res;
  }
  return *this;
}

main() {
  String s {"Init"};
  (s-"ni").print();
}
```



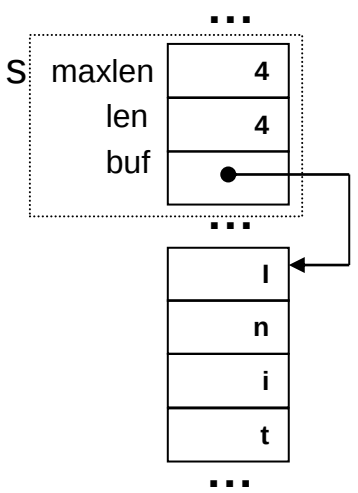
Zugriff auf bereits freigegebenen Speicher!

Temporäres Objekt

Weitere Probleme mit Klasse String

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

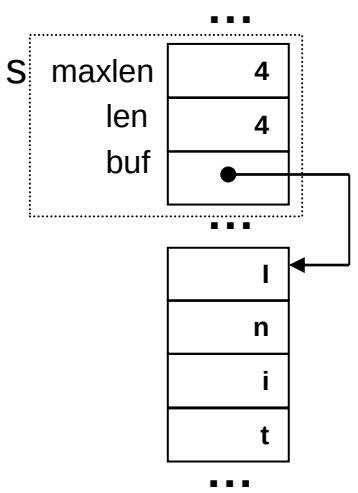


Weitere Probleme mit Klasse String

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

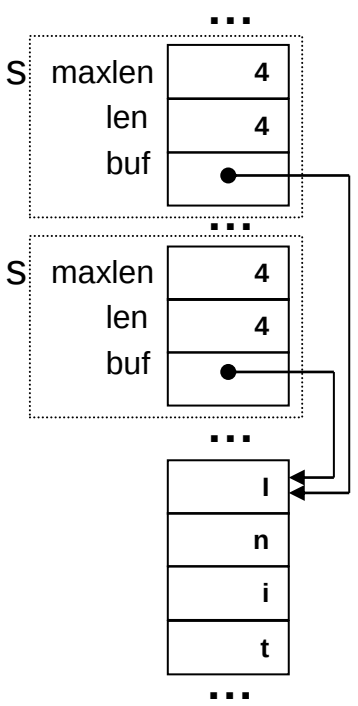
Call by value kopiert den aktuellen Parameter in den formalen



Weitere Probleme mit Klasse String

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

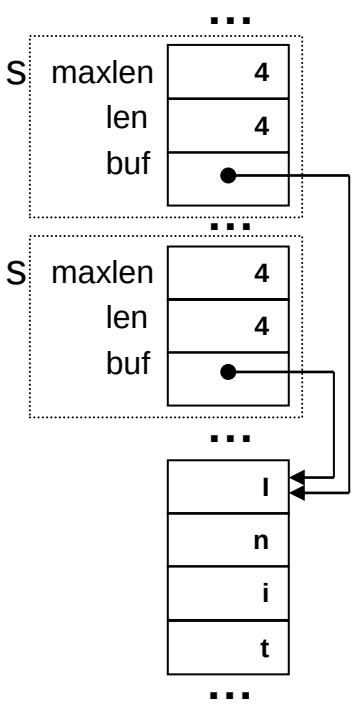


Weitere Probleme mit Klasse String

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

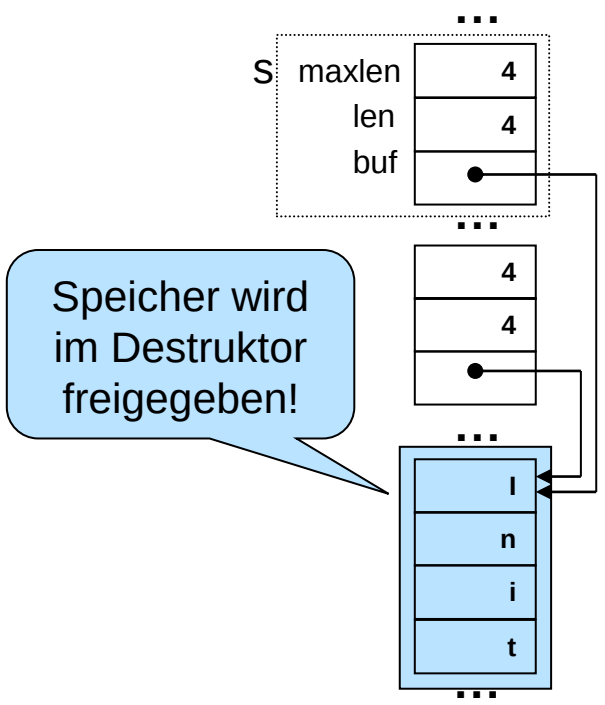
Destruktoraufruf für Parameter



Weitere Probleme mit Klasse String

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

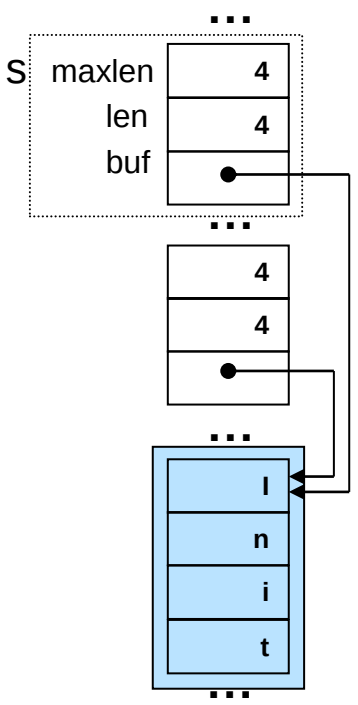


Weitere Probleme mit Klasse String

```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```

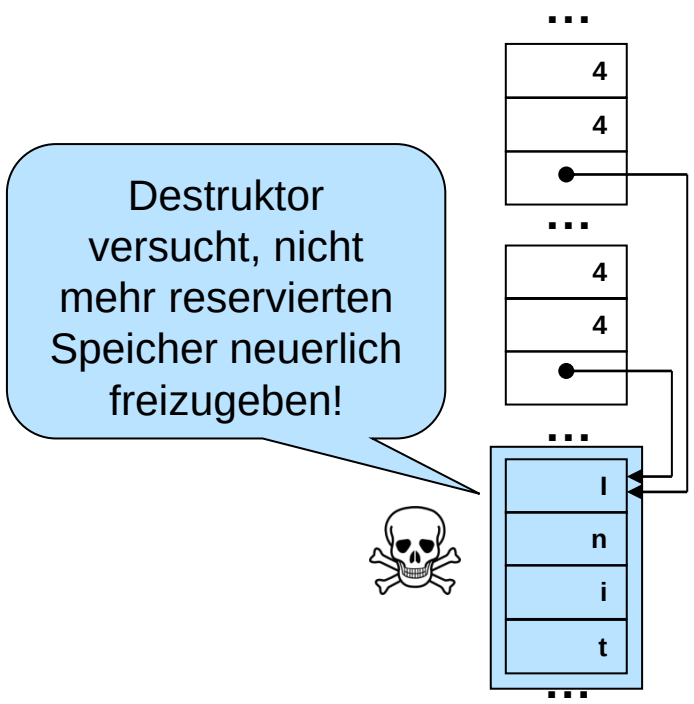
Destruktoraufruf für lokale Variable



Weitere Probleme mit Klasse String

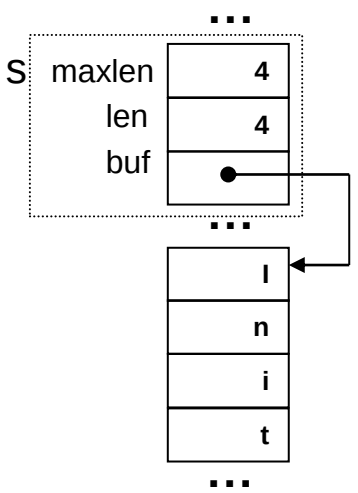
```
void foo(String s) {
}

main() {
  String s {"Init"};
  foo(s);
}
```



Weitere Probleme mit Klasse String

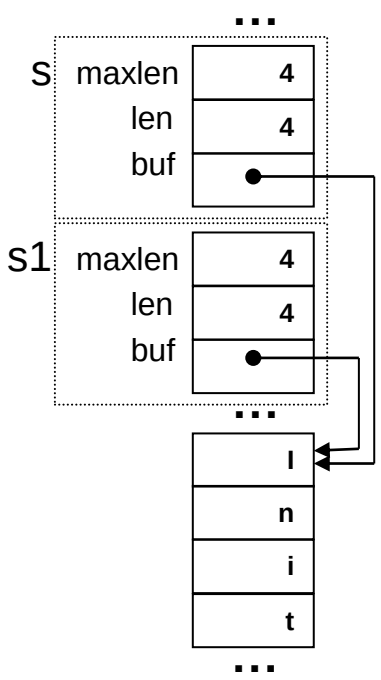
```
main() {
  String s {"Init"};
  String s1 {s};
}
```



Weitere Probleme mit Klasse String

```
main() {
  String s {"Init"};
  String s1 {s};
}
```

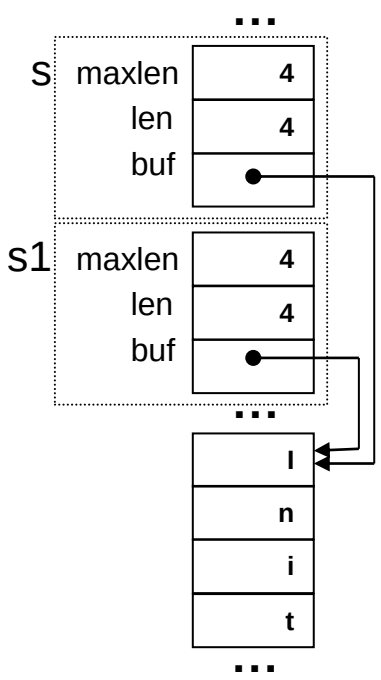
Initialisierung mittels komponentenweiser Kopie!



Weitere Probleme mit Klasse String

```
main() {
  String s {"Init"};
  String s1 {s};
}
```

Zerstörung der lokalen Variablen

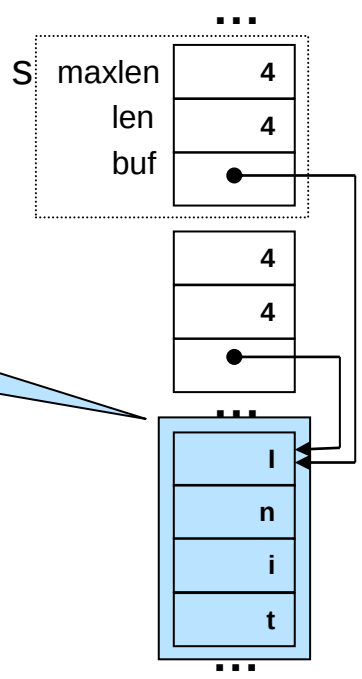


Weitere Probleme mit Klasse String

```
main() {
  String s {"Init"};
  String s1 {s};
}
```

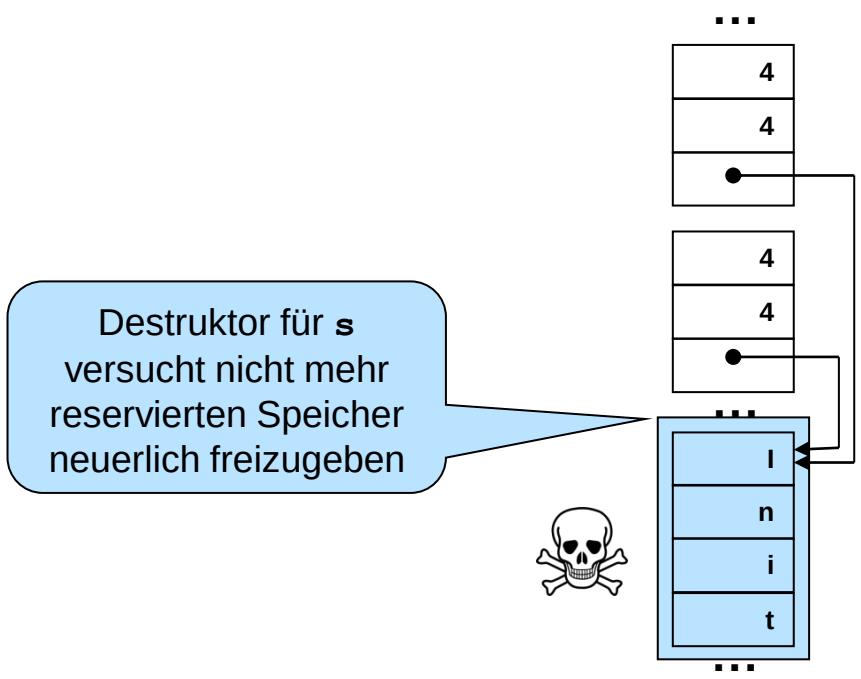
Zerstörung der lokalen Variablen

Destruktor für **`s1`** gibt Speicher frei



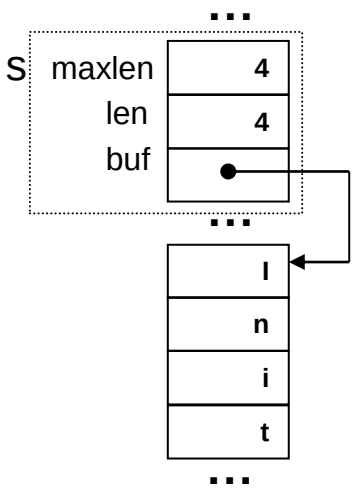
Weitere Probleme mit Klasse String

```
main() {
  String s {"Init"};
  String s1 {s};
}
```



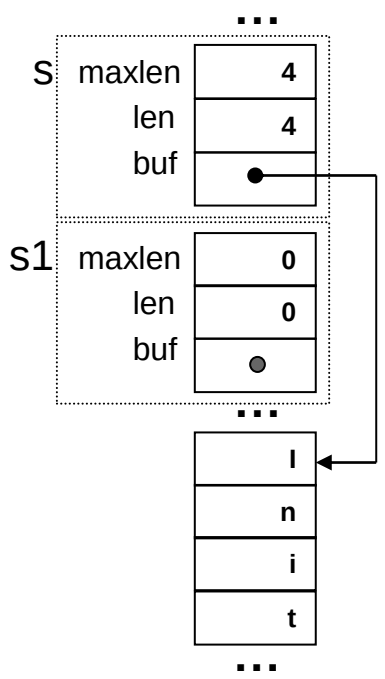
Noch immer nicht alle Probleme der Klasse String gelöst

```
main() {
  String s {"Init"};
  String s1;
  s1 = s;
}
```



Noch immer nicht alle Probleme der Klasse String gelöst

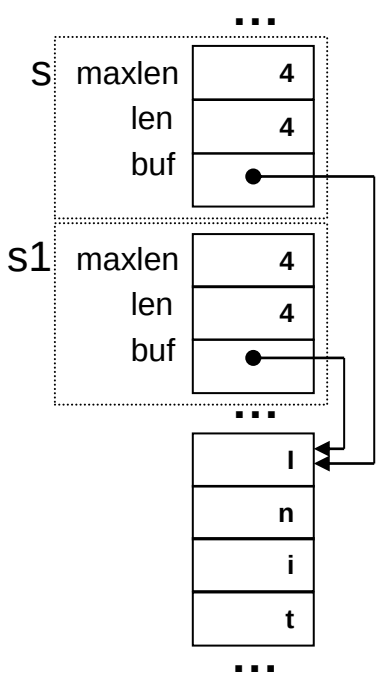
```
main() {
  String s {"Init"};
  String s1;
  s1 = s;
}
```



Noch immer nicht alle Probleme der Klasse String gelöst

```
main() {
  String s {"Init"};
  String s1;
  s1 = s;
}
```

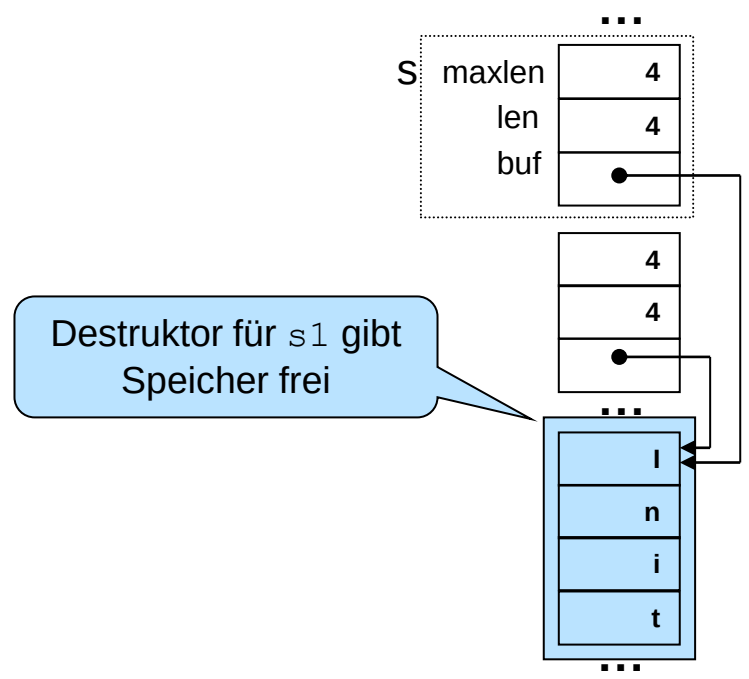
Keine Initialisierung!
Keine Kopie!
Komponentenweise Zuweisung



Noch immer nicht alle Probleme der Klasse String gelöst

```
main() {
  String s {"Init"};
  String s1;
  s1 = s;
}
```

Zerstörung der lokalen Variablen



Destruktor für `s1` gibt Speicher frei

Noch immer nicht alle Probleme der Klasse String gelöst

```
main() {
  String s {"Init"};
  String s1;
  s1 = s;
}
```