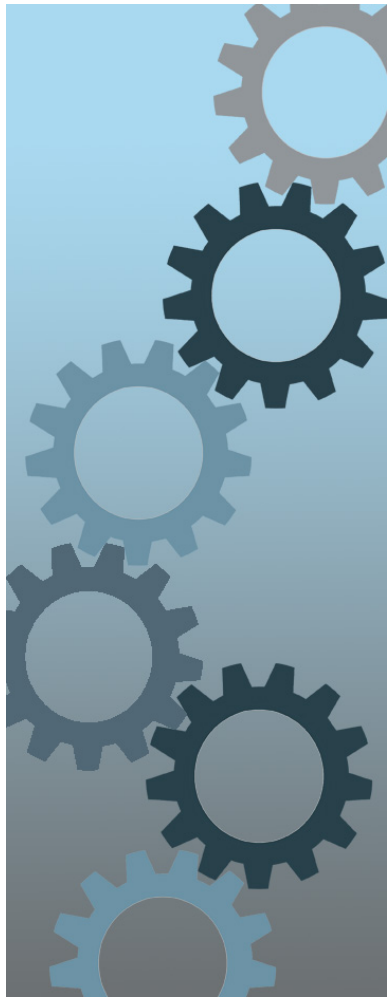
Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Programming Concepts & Paradigms

Imperative Sprachen am Beispiel C & Java



Prof. Dr. Ruedi Arnold

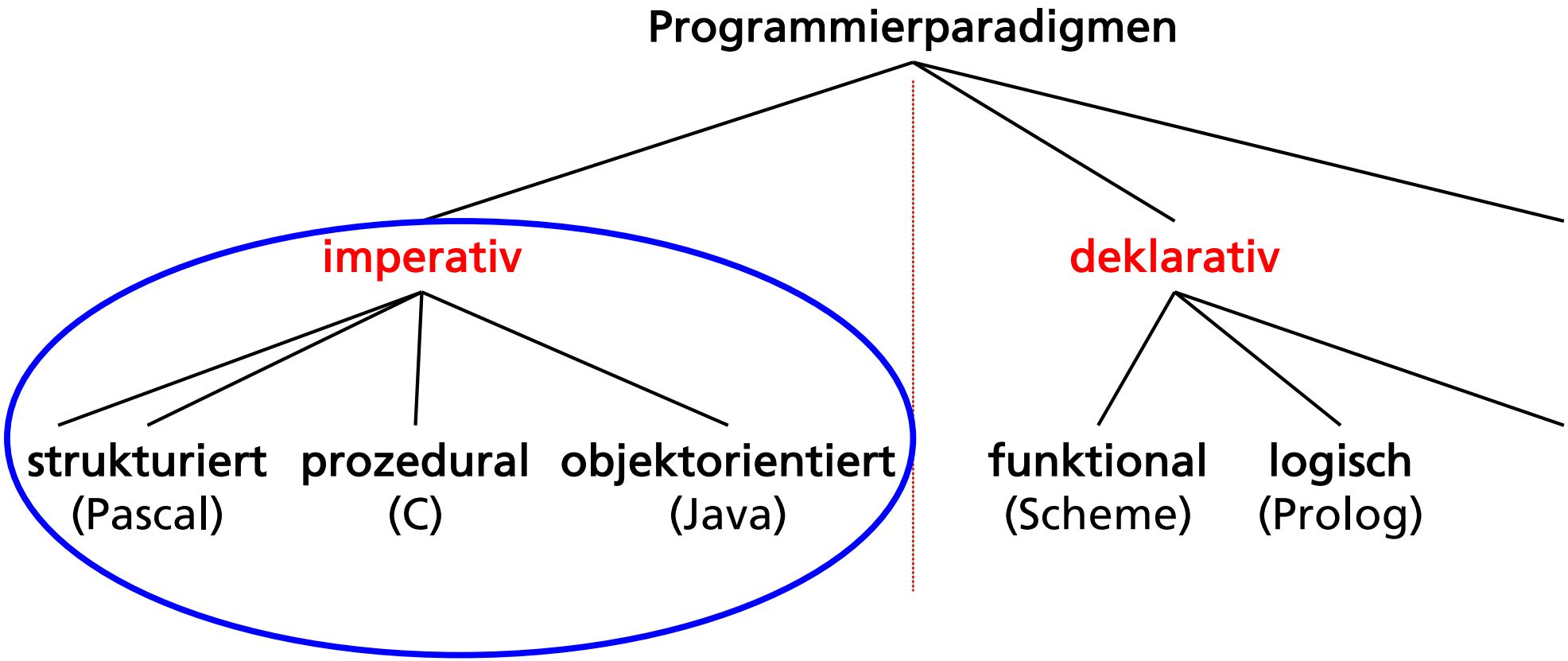
ruedi.arnold@hslu.ch

FH Zentralschweiz

Übersicht: Imperative Sprachen am Bsp. C & Java

- Die Programmiersprachen C und Java
  - imperativ
  - strukturiert
  - prozedural
- Java: Objektorientierung
  - Abstrakter Datentyp vs. Klasse
  - Zeiger (C) vs. Referenz (Java)

Wiederholung: Paradigmen & typische Sprache



➔Heute im Fokus: Imperative Paradigmen (strukturiert, prozedural, objektorientiert) am Beispiel C & Java

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

C & Java

Einleitung: C & Java

- Am Beispiel der Programmiersprachen C und Java untersuchen wir ausgewählte Aspekte von imperativer Programmierung
- Beide Sprachen erlauben imperative, strukturierte und prozedurale Programmierung
  - Java erlaubt zusätzlich objektorientierte Programmierung, mehr dazu später
- Beide Sprachen sollten bekannt sein
  - Java auf jeden Fall vom HSLU-Studium ☺
  - C evtl. vom HSLU-Studium...
    - Don‘t panic: notwendiges C-Wissen wird erläutert ☺

Java ist stark von C beeinflusst

- Java basiert stark auf C - Syntax und Semantik sind weitgehend identisch
  - Viele gleiche Schlüsselwörter wie: char, double, else, enum, float, for, goto, if, int, long, return, short, static, switch, void, while
  - Weitgehende gleiche Klammerung von Blöcken mittels geschweifter Klammern: { /* Block */ }
  - Gleiche Syntax und Semantik für die zentralen Kontrollstrukturen: if, if-else, for- und while-Schleifen
  - Weitgehend gleiche elementare Datentypen, wie z.B. int, char, float
  - usw.

Alle C- und Java-Schlüsselwörter

C exklusiv	C & Java	Java exklusiv
auto, _Bool, _Complex, extern, _Imaginary, inline, register, restrict, signed, sizeof, struct, typedef, union, unsigned	break, case, char, const*, continue, default, do, double, else, enum, float, for, goto*, if, int, long, return, short, static, switch, void, volatile, while	abstract, assert, boolean, byte, catch, class, extends, final, finally, implements, import, instanceof, interface, native, new, package, private, protected, public, strictfp, super, synchronized, this, throw, throws, transient, try

* reserved, but not used in Java

Quellen:

- https://en.wikipedia.org/wiki/C_syntax#Reserved_keywords
- http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Wiederholung: Imperative Programmierung

- lat.: imperare = befehlen, anordnen
- Problem-Lösungs-Algorithmus wird schrittweise durch Befehle angegeben: Der Quellcode gibt an, was der Computer in welcher Reihenfolge tun soll
- Zur Steuerung der Befehlsausführung stehen Kontrollstrukturen (Sequenz, Schleifen, Verzweigung) zur Verfügung
- So funktionieren die meisten bekannten und populären Programmiersprachen wie z.B. Java, C#, PHP, usw.
  - Insbesondere funktionieren Hardware-nahe Sprachen wie Assembler (oder C) praktisch immer so

Imperative Programmierung: Grundidee (Pseudocode)

```
„Startpunkt“                    % Vorgegeben (typischerweise „main“ o.ä.)
1.  mach A                      % Anweisung A
2.  falls B mach C              % Verzeigung, in C: if-Anweisung
3.  mach D                      % Anweisung D
4.  wenn E springe zu 3.        % Verzeigung & Sprung, in C: „if + goto“
5.  mach F                      % Anweisung F
```

- Programm besteht aus Sequenz von Anweisungen
- Kontrollfluss ist klar vorgegeben, grundsätzlich sequentiell
- Typische Kontrollstrukturen imperativer Programmierung
  - Verzweigung (if)
  - Sprung (goto)
  - Schleifen (while, oder gleichmächtig: for)
  - Falls Prozeduren: Prozedurale Programmierung

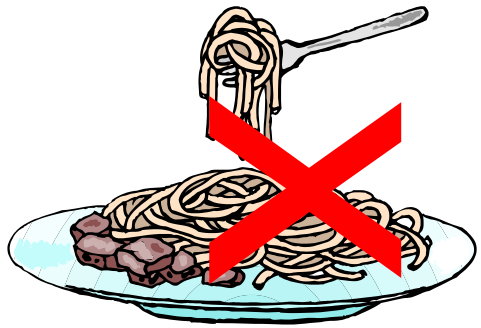
Beispiel: Imperatives C-Programm, inkl. „goto“

```
int a = 25;                         % Variablen-Deklaration
int b = 15;                         % Variablen-Deklaration
start:                              % Sprungmarke
if (a > b) a = a - b;               % Auswahl
else b = b - a;                     % Alternative
if (a != b) goto start;             % Bedingung
printf("result = %i", a);           % Konsolen-Ausgabe
```

- Was gibt diese kleine C-Programm-Sequenz aus?

Wiederholung: Strukturierte Programmierung

- Spezialisierung des imperativen Paradigmas, verlangt Beschränkung auf drei Kontrollstrukturen:
  1. Sequenzen (Hintereinander auszuführende Programmanweisungen)
  2. Auswahl (Verzweigung: Bedingung)
  3. Wiederholung (Schleifen)
  - Konsequenz: „goto“ darf nicht eingesetzt werden, sonst „Spaghetticode“ (= Code mit verworrenen Kontrollstrukturen)
    - Artikel "Go To Statement Considered Harmful" (1968) von Edsger Dijkstra war Wendepunkt/Auslöser
- Typische Sprachen: C, Fortran, Pascal



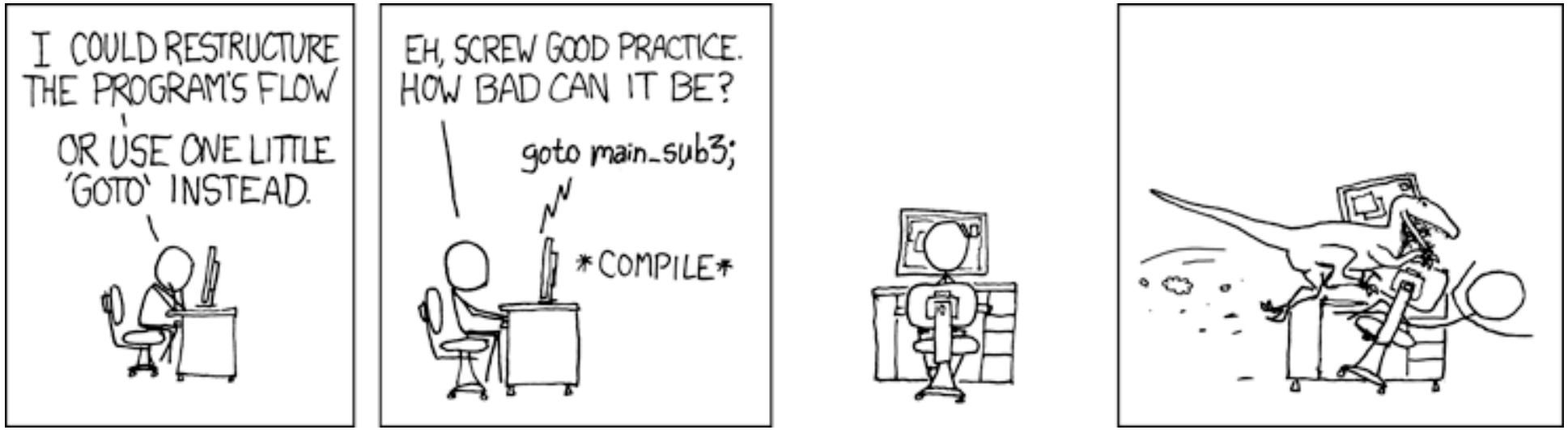
Strukturierte Programmierung: Kein „goto“

- Relevante Einschränkung: unbedingter Sprung (goto-Anweisung) darf nicht verwendet werden
- Motivation: mit der goto-Anweisung sind verworrene Kontrollstrukturen möglich (= „Spaghetticode“)
  - Auswahl und Wiederholung sind einfacher verständlicher und Programmfluss nachvollziehbar

„Go To Statement Considered Harmful“

- Lesenswerter Artikel von Edsger Dijkstra
  - Com. of the ACM, Vol. 11, No. 3, March 1968
    - online Version:
      http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html
  - Argumentation über Charakterisierung des Prozess-Fortschritts: Wo sind wir im Programmablauf?
  - Interessante Nebenbemerkungen im Artikel:
    - Schleifen sind grundsätzlich überflüssig (es gibt ja rekursive Prozeduren), aber „praktisch“
    - Jemand hat offensichtlich bewiesen, dass goto theoretisch überflüssig ist
      - Code mit goto kann mechanisch in ein (ggf. sehr intransparentes, daher nicht zum empfehlen!) Flussdiagramm ohne goto umgewandelt werden

...damit Sie mir auch glauben, dass goto böse ist und nicht verwendet werden sollte:



https://imgs.xkcd.com/comics/goto.png

;-)

Beispiel: strukturiertes C-Programm

```
int a = 25;                          % Variablen-Deklaration
int b = 15;                          % Variablen-Deklaration
while (a != b) {                     % Schleife
  if (a > b) a = a - b;              % Auswahl
  else b = b - a;                    % Alternative
}
printf("result = %i", a);            % Konsolen-Ausgabe
```

- Was gibt diese kleine C-Programm-Sequenz aus?

Wiederholung: Prozedurale Programmierung

- Spezialisierung des imperativen Paradigmas, bietet Unterteilung von Programmen in Teilprogramme (= Prozedur, auch: Routine, Unterprogramm, Funktion)
  - Vermeidet Code-Duplikation
  - Prozeduren können Argumente entgegen nehmen und Ergebnisse zurück geben
  - Unterscheidung in Programm-globale und Prozedur-lokale Variablen
- Oft genannt als „Gegenstück“ zur objektorientierten Programmierung innerhalb der imperativen Sprachen
- Typische Sprachen: C, Fortran, Pascal

Beispiel: prozedurales C-Programm

```
int main() {                                % Hauptfunktion
    int result = gcd(25, 15);               % Funktionsaufruf
    printf("result = %i", result);          % Konsolen-Ausgabe
    return 0;                               % Funktionsrückgabe
}

int gcd(int a, int b) {                     % Funktionskopf
    while (a != b) {                        % Schleife
        if (a > b) a = a - b;               % Auswahl
        else b = b - a;                     % Alternative
    }
    return a;                               % Funktionsrückgabe
}
```

Kontrollfragen A

1. Charakterisieren Sie in eigenen Worten, wie imperative Programmierung funktioniert. Aus welchen Grundelementen sind imperative Programme aufgebaut? Wie werden derartige Programme ausgeführt, wie verläuft der Kontrollfluss?
2. In welchem Verhältnis stehen imperative und strukturierte Programmierung?
3. Durch was hebt sich die prozedurale Programmierung vom imperativen Programmierparadigma ab?

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Datentypen in C

Datentypen in C

- Bisher gesehen: C kennt elementare Datentypen wie int, char oder double und die Datenstruktur Array[]
- Frage: Was ist mit komplexeren (eigenen, zusammengesetzten) Datentypen wie Rechteck oder Kunde und mit Datenstrukturen wie Stack oder Liste?
- Antwort: In C können zusammengesetzte Datentypen und Datenstrukturen mit Hilfe von Strukturen (struct) und einer Typdefinition (typedef) erzeugt werden

C: Strukturen (struct)

- Strukturen fassen eine Liste von Variablen zusammen
- Schlüsselwort `struct`, allgemeine Form:

```
struct struct_name {
   type1 member1;
   type2 member2;
   // and so on...
};
```

- Konkretes Beispiel für eine Struktur „Punkt“ mit x- und y-Koordinaten und einer Farbe (als Charakter kodiert)

```
struct point {
    int x;
    int y;
    char color;
};
```

```
struct point p;
p.x = 100;
p.y = 200;
p.color = 'g';
printf(" p = (%i, %i)", p.x, p.y);
```

C: eigene Typen definieren mittels typedef

- `typedef`: eigene Datentypen definieren
  - Einfaches Anwendungsbeispiel: Wrapper-Typen für die Datentypen `int` und `char`

```
typedef int people_count;           // create own data type
typedef char one_letter;            // create own data type
people_count attendees = 55;        // use own data type
one_letter the_letter = 'p';        // use own data type
```

- Einsatzzweck: gemäss den beiden C-Erfindern Brian Kernighan & Dennis Ritchie ermöglicht `typedef`...
  - ...einfacher portierbaren Code
  - ...verständlicheren Code

struct & typedef: Struktur als eigener Datentypen

- `struct` und `typedef`: oft in Kombination verwendet, um eigene Datentypen für Datenstrukturen zu definieren
- Anwendungsbeispiel: eigener Datentyp für Punkte

```
typedef struct {
    int x;
    int y;
    char color;
} point_type;
```

```
point_type pt;
pt.x = 100;
pt.y = 200;
```

  - Hinweis: Im Gegensatz zur Verwendung von `struct`-Typen (ohne `typedef`) ist hier bei der Deklaration einer Variable das Schlüsselwort `struct` nicht nötig, siehe z.B.:

```
struct point p;
...
```

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Abstrakte Datentypen

(inkl. Bsp. Stack)

C: Abstrakte Datentypen (ADT)

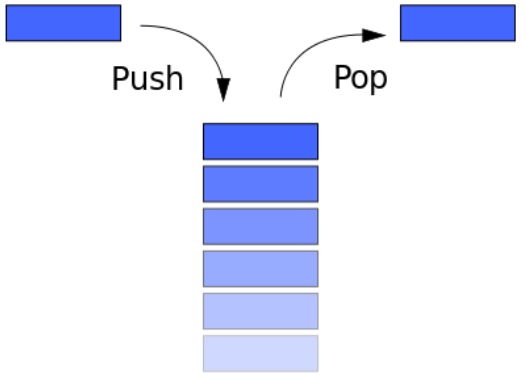
- Abstrakte Datentypen (ADT) sind mathematische Modelle für Datenstrukturen und darauf definierte Operationen (mit vorgegebenem Effekt)
  - Semantik ist vorgegeben, nicht aber Implementierung

- Typische Beispiele für ADTs

ADT Name	Operationen
Stack	init, push, pop, top, (isEmpty)
Queue	init, add, remove, (isEmpty)
Matrix	standard matrix operations: addition, mulitplication, inverse, ...

ADT-Beispiel: Stack

- Stack = Sammlung von Elementen mit bestimmter Struktur: Neue Elemente könnten hinzugefügt werden (Operation **`push`**), das zuletzt hinzugefügte Element kann abgefragt werden (**`top`**) und mit **`pop`** vom Stack entfernt werden.
  - LIFO-Prinzip: Last-in-first-out
  - Semantik der Stack-Operationen
    - init: liefert leeren Stack zurück
    - push: fügt ein Element zum Stack hinzu
    - top: liefert das zuletzt hinzugefügte Element zurück
    - pop: entfernt das zuletzt hinzugefügte Element
    - print: Gibt den aktuellen Stack aus, ohne den Stack zu verändern (Operation zu Testzwecken hinzugefügt)



http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29#mediaviewer/File:Data_stack.svg

ADT Stack: Funktionssignaturen & Semantik

- Generische Signaturen Stack-Operationen:

```
stack init();                          // returns new empty stack
stack push(element e, stack s);        // adds element to stack
element top(stack s);                  // returns top element
stack pop(stack s);                    // removes top element
void print(stack s);                   // prints all elements
```

- Semantik der Stack-Operationen:

```
top(init()) = ERROR
top(push(e, init()) = e
pop(init()) = init()
pop(push(e, s)) = s
```

  - Wichtig: Diese Semantik ist unabhängig von der Implementierung!
  - (Hinweis: print-Operation verändert Stack nicht, ist daher für die Semantik irrelevant)

Einfache Beispiel-Implementierung: Stack in C

- Elemente-Typ: int, verpackt als Datentyp element

```
typedef int element;
```

- Eigener Datentyp stack (mittels typedef & struct)

```
#define STACK_SIZE 3
typedef struct {
    element stackArray[STACK_SIZE];
    int index;
} stack;
```

  - Diese Stack-Implementierung besteht also aus Array von Elementen und Index auf den nächsten freien Platz

- Implementierung entsprechender C-Funktionen: siehe Code im Git-Repo, bzw. auf folgenden Folien...

Einfache Stack-Impl.: init()

```
1.  #define STACK_SIZE 3
2.  #define STACK_EMPTY_INDEX -1
3.  #define STACK_DUMMY_ELEMENT -1234 // arbitrary number

4.  // returns new empty stack
5.  stack init() {
6.      stack s;
7.      s.index = STACK_EMPTY_INDEX;
8.      return s;
9.  }
```

- Stack wird angelegt (6) und der Index initalisiert (7)

Einfache Stack-Impl.: push(...)

```
1.   stack push(element e, stack s) {
2.       if (s.index + 1 < STACK_SIZE) {
3.           // there is space for one more element
4.           s.index++;
5.           s.stackArray[s.index] = e;
6.       } else {
7.           printf("ERROR - push: stack full! Cannot add %i\n", e);
8.       }
9.       return s;
10.  }
```

- Falls Platz im Array (2) wird Index hochgezählt (4) und Element eingefügt an hochgezählte Index-Position (5)
- Falls kein Platz (6): Fehlermeldung in Konsole (7)
  - D.h.: ganz elementare Fehlerbehandlung!
- Stack wird auf jeden Fall wieder zurückgeliefert (9)

Einfache Stack-Impl.: top(...)

```
1.   element top(stack s) {
2.       if (s.index > STACK_EMPTY_INDEX) {
3.           return s.stackArray[s.index];
4.       } else {
5.           printf("ERROR - top: stack empty!\n");
6.           return STACK_DUMMY_ELEMENT;
7.       }
8.   }
```

- Falls Array nicht leer (2) wird das letzte Element zurückgeliefert (3)
- Falls Array leer: Ausgabe Fehlermeldung (5) und Rückgabe Dummy-Element (6)
  - D.h.: ganz simple Fehlerbehandlung! (Könnte/sollte natürlich besser/aufwändiger gemacht werden)

Einfache Stack-Impl.: pop(...) und print(...)

```
1.   stack pop(stack s) {
2.       if (s.index > STACK_EMPTY_INDEX) {
3.           s.index--;
4.       }
5.       return s;
6.   }
7.
8.   void print(stack s) {
9.       if (s.index > STACK_EMPTY_INDEX) {
10.          printf("Stack contains: ");
11.          int i;
12.          for (i = 0; i <= s.index; i++) {
13.              printf("%i, ", s.stackArray[i]);
14.          }
15.          printf("top element = %i\n", s.stackArray[s.index]);
16.      } else {
17.          printf("print - Stack is empty\n");
18.        }
19.  }
```

- Code sollte verständlich sein – Fragen?

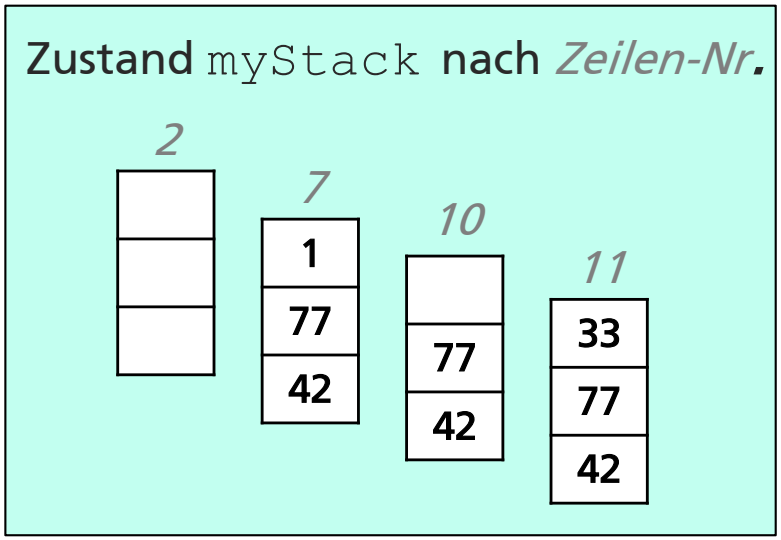
Funktionsargument struct: call by value (vs. reference)

- Funktionssignatur von pop: `stack pop(stack s)`
  - Gesehen: Datentyp stack ist definiert als `struct`
- Wichtig: Strukturen (`struct`) werden in C auf diese Art als Werte (d.h. Kopie) übergeben: call by value

- Bemerkung: Strukturen können in C auch als Referenzen übergeben werden
  - Dazu müsste mit Referenzen (Zeiger, Pointers) auf Strukturen gearbeitet werden, das schauen wir hier jedoch nicht weiter an
    - Hinweis: Auch diese Zeiger werden dann aber als Werte übergeben, d.h. auch hier bleibt das ein „call by value“, aber im Gegensatz zu oben kann der Wert (also die struct) hinter der Referenz verändert werden

Einfache Stack-Impl.: Anwendung

- Die main-Methode →




```
1.  int main(int argc, char** argv) {
2.      stack myStack = init();
3.      print(myStack);
4.      top(myStack);
5.      myStack = push(42, myStack);
6.      myStack = push(77, myStack);
7.      myStack = push(1, myStack);
8.      print(myStack);
9.      myStack = push(33, myStack);
10.     myStack = pop(myStack);
11.     myStack = push(33, myStack);
12.     print(myStack);
13.     element e = top(myStack);
14.     printf("top element is %i\n", e);
15.     return (EXIT_SUCCESS);
16. }
```

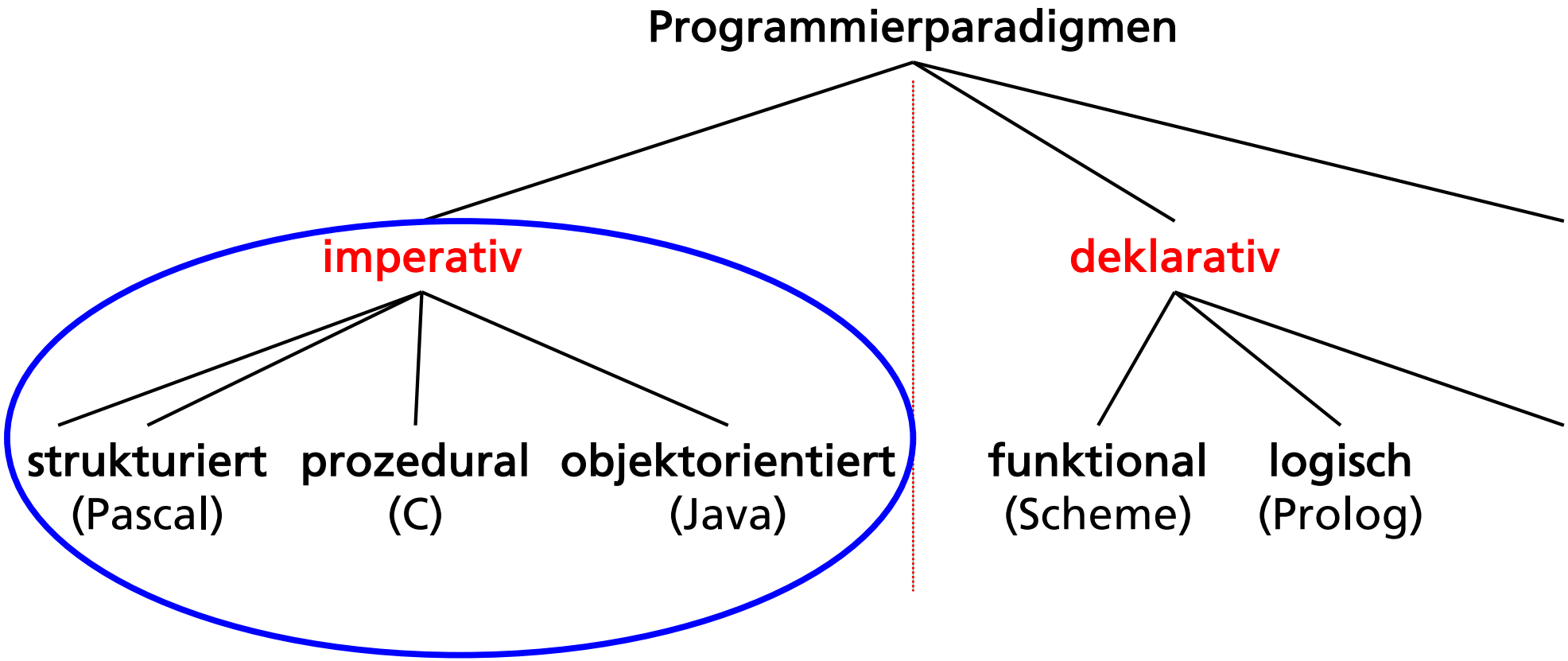
- Ausgabe Konsole:

```
1.  print - Stack is empty
2.  ERROR - top: stack empty!
3.  print - Stack contains: 42, 77, 1, top element = 1
4.  ERROR - push: stack full! Cannot add 33
5.  print - Stack contains: 42, 77, 33, top element = 33
6.  top element is 33
```

Bem. zu einfacher Stack-Implementierung in C

- Als unterliegende Datenstruktur wäre statt statischem Array z.B. auch dynamisch verkettete Liste möglich
  - Hier nicht gewählt, da Array einfacher und für Liste z.B. Zeiger-Typen notwendig wären
- Statt `int` als Element-Typ wäre praktisch beliebiger anderer Typ möglich
  - `int` gewählt der Einfachheit halber, verpackt in Typ `element` mithilfe von `typedef`

Wiederholung: Paradigmen & typische Sprache



➔ Jetzt schauen wir an, wie sich objektorientierte Sprachen aus prozeduralen Sprachen entwickelten, resp. worin sich diese beiden Paradigmen unterscheiden

Von ADT (prozedural) zu Klassen (OO)...

- ADT bieten also Datenstrukturen und darauf klar definierte Operationen (resp. Funktionen)
  - „Unschönheit“: Implementierung der Datenstruktur ist (grundsätzlich) getrennt und unabhängig von der Implementierung der dazu gehörigen Funktionen
- Zusammenbringen von Daten und Operationen war eine der Hauptmotivationen für „Objektorientierung“
  - Die Funktionen „hängen“ direkt an den Daten: Klassen haben definierte „Daten“ (Zustand: Instanzvariablen) und „Funktionen“ (Verhalten: Methoden)

➔ OO ist in diesem Sinn eine Weiterentwicklung von ADTs ☺

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

OO

Wiederholung: Objektorientierte Programmierung

- Spezialisierung des imperativen Paradigmas, laufende Programme bestehen aus einzelnen Objekte, welche miteinander interagieren (Nachrichten austauschen = Methoden aufrufen)
  - So wie sie objektorientierte Programmierung an der HSLU anhand von Java in PRG1, OOP, usw. gelernt haben
- Objekte sind typischerweise Instanzen von Klassen (geht auch anders, z.B. mit Prototypen in JavaScript)
  - Klassen definieren Zustand (Variablen) und Verhalten (Methoden)
  - Vererbung, Polymorphismus, usw.
- Entstanden als Weiterentwicklung von prozeduraler Programmierung und abstrakten Datentypen
- Typische Sprachen: Smalltalk, Objective C, C++, Java, C#

OO: Packages und Sichtbarkeiten

- Mit der Definition von Klassen und Packages (hierarchischen Gruppen von Klassen) wurden auch Fragen der Sichtbarkeit relevant(er): Von wo aus kann auf was (Methode / Variable) zugegriffen werden?
- Lösung von Java kennen sie: vierstufige Sichtbarkeit
  - Deklariert durch die vier Zugriffsmodifizierer
    - `private`
    - *`[default]`*
    - `protected`
    - `public`

Java „extends“ C

- Java erweitert C also im wesentlichen um folgende Konzepte (und `Schlüsselwörter`)
  - Klassen + Instanzen: `class`, `extends`, `instanceof`, `new`, `this`, `super`
  - Interfaces: `interface`, `implements`
  - Packages: `package`, `import`
  - Sichtbarkeit: `private`, `protected`, `public`
  - Exception-Handling: `try`, `catch`, `finally`, `throw(s)`

Wichtiger Unterschied: C-Zeiger vs. Java-Referenzen

- C verwendet Zeiger (EN: pointer)
  - d.h. direkte Speicheradressierung, z.B.

```
int a = 42;                     // declaration & init of int
int *pa = &a;                   // pa is a pointer to a int
*pa = 77;                       // change the value pa is
                                // pointing to to 77
printf("a is %i", *pa);         // prints "a is 77"
```

    - `&x`: Referenzierung, liefert Speicheradresse der Variablen x
    - `*y`: Dereferenzierung, liefert den Wert an der Speicheradresse der Zeigervariablen y
- In Java spricht man grundsätzlich nicht von Zeigern, sondern von Referenzen, entsprechend gibt's in Java z.B. auch keine Zeigerarithmetik wie in C

Bekannte Ausnahme:
`Null`**`Pointer`**`Exception` ;-)

Stack: Einfache Java-Implementierung

- In Java können die Daten und die Operationen des ADT Stack in einer Klasse zusammengefasst werden
  - Stack-Operationen können mit Hilfe entsprechender Instanz-Methoden direkt auf Stack-Instanzen aufgerufen werden, z.B.:

```
1.  Stack myStack = new Stack();
2.  myStack.print();
3.  myStack.top();
4.  myStack.push(new Element(42));
```

  - Anstelle vom davor gesehenen ADT-Stil in C:

```
1.  stack myStack = init();
2.  print(myStack);
3.  top(myStack);
4.  myStack = push(42, myStack);
```

Übung 1: einfacher Stack in C und Java

- Dieser Java-Code:

```
1.  Stack myStack = new Stack();
2.  myStack.print();
3.  myStack.top();
4.  myStack.push(new Element(42));
5.  myStack.push(new Element(77));
6.  myStack.push(new Element(1));
7.  Element e = myStack.top();
8.  System.out.println("top Element is " + e.getValue());
9.  myStack.push(new Element(33));
10. myStack.print();
```

...soll beispielsweise diese Konsolen-Ausgabe produzieren:

```
1.  print - Stack is empty
2.  ERROR - top: stack empty
3.  top Element is 1
4.  print - Stack contains: 33, 1, 77, 42, top Element = 33
```

Übung 1: C-Code im GitLab

- Stack-Implementierung in C ist im GitLab verfügbar
  - 3 Dateien
    - **stack.h:** Header-Datei mit Deklaration der Stack-Funktionen, Stack-Datentyp und Konstanten
    - **stack.c:** Stack-Implementierung basierend auf einem Array
    - **main.c:** main-Funktion, welche den Stack verwendet
- Verwenden und erweitern sie diesen Code

Hinweis: PCP-Code @ GitLab

- PCP-Code geben wir via GitLab ab
  - z.B. Code von Folien oder Fragmente für Übungen...

➔https://gitlab.enterpriselab.ch/PCP/PCP-public-Code

  - Klappt Zugriff überall? ☺

Kontrollfragen B

1. Was ist ein ADT?
2. Worin unterscheidet sich die Implementierung von einem ADT in C fundamental von einer Implementierung in Java?
3. Um welche Konzepte erweitert die OO-Programmiersprache Java im wesentlichen die prozedurale Programmiersprache C?

Abschlussbemerkungen

- Java hängt stark von C ab
  - Java ergänzt C primär um OO-Konzepte
- OO kann als Weiterentwicklung von prozeduraler Programmierung und ADTs gesehen werden
  - Klassen in C++ sind Erweiterungen von structs
- Damit sollten sie die Programmierparadigmen imperativ, strukturiert, prozedural und objektorientiert einordnen und gegeneinander abgrenzen können

...und nächste Woche geht's los mit deklarativ-logischer Programmierung in Prolog!