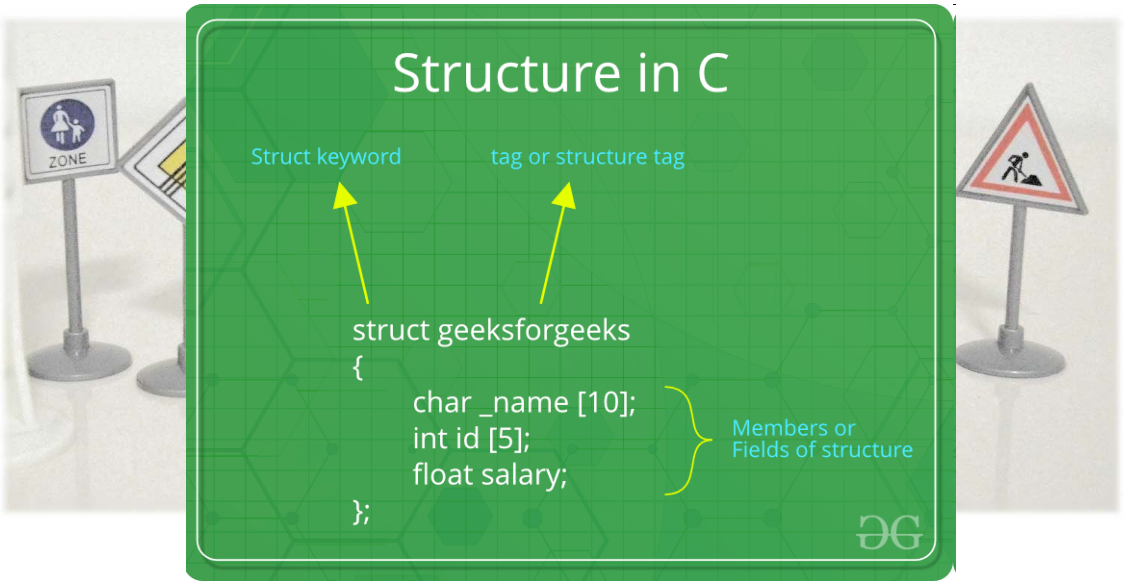
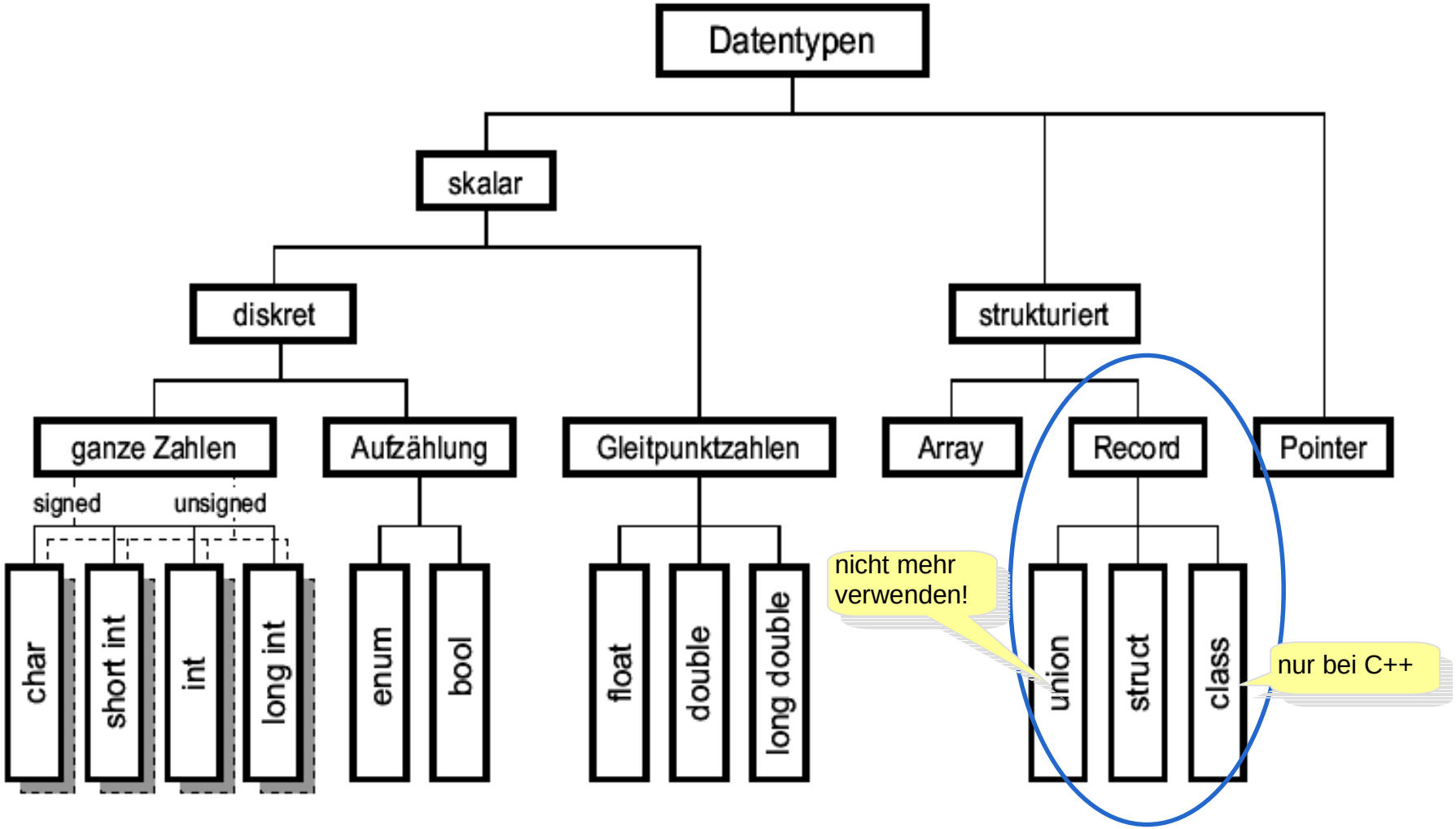
INF1
Datenstrukturen



- Datenstrukturen mit struct
- Zeiger auf Datenstrukturen
- Verkettete Strukturen
- Make File

Datenstrukturen mit struct

Datentypen Übersicht



Structs

- Mehrere Variablen zu einer Einheit zusammenfassen
- Im Unterschied zu Arrays kann eine Struktur aus Elementen verschiedener Datentypen zusammengesetzt sein

```
struct Complex {
    double real;
    double imag;
};
struct Complex x;
```

Semikolon nicht vergessen!

- Auf die Werte des Structs greift man mit der "." Notation zu

```
x.real = 7.0;
x.imag = 6.4;
```

Structs – weitere Beispiele

- Definition des Structs und Variablendeklaration kombiniert

```
struct Complex {
    double real;
    double imag;
} x;                    /* inklusive Variablendefinition */
```

Structs – weitere Beispiele

- Es können auch mehrere Variablen vom Struct Typ definiert werden
- Bei der Definition können sie auch gleich initialisert werden

```
struct Datum {
    int tag;
    int monat;
    int jahr;
};

struct Datum meinGeburtstag = {17, 4, 1976};
struct Datum geburtstag;
```

Initialisierung der Werte

Zuweisung

- Strukturvariablen können einander zugewiesen werden
- Dabei werden die/alle Werte kopiert: Unterschied zu Arrays !

```
struct Datum meinGeburtstag = {17, 4, 1976};

struct Datum geburtstag;



geburtstag = meinGeburtstag ;

geburtstag.tag = 25;
printf("%d", meinGeburtstag.tag);    // 17
printf("%d", geburtstag.tag);        // 25
```

Typedef

Typedef

- typedef ist ein Schlüsselwort, das zur Erstellung eines Alias für einen Datentyp verwendet wird.

Allgemeine Form: **`typedef <existing_name> <alias_name>`**

```
typedef unsigned long ulong;
typedef int* IntPtr;
IntPtr px, py, pz;
```

… Typedef

- Typen mit Plattform unabhängig definierter Grösse in <stdint.h>

```
/* Exact-width integer types */
typedef signed char int8_t;
typedef unsigned char   uint8_t;
typedef short  int16_t;
typedef unsigned short  uint16_t;
typedef int  int32_t;
typedef unsigned   uint32_t;
typedef long long  int64_t;
typedef unsigned long long   uint64_t;
```

http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/030_c_anhang_b_017.htm

Struct Definition mit Typedef

- Ein Strukt kann auch mit *typedef* angelegt (definiert) werden

```
typedef struct {
    int tag;
    int monat;
    int jahr;
} Datum;

Datum variable;
```

Hier kein "strukt" mehr nötig!

Abwägen:
Es ist elegant, einfach *NeuerTyp variable;* schreiben zu können.
Auf der anderen Seite wird mit *struct NeuerTyp variable;* klarer, dass es sich um eine Datenstruktur handelt...

Zeiger auf Datenstrukturen

Zeiger auf Strukturen

```
typedef struct {
     int tag;
     int monat;
     int jahr;
 } Datum;

 Datum geburtstag;
 Datum *pgb = &geburtstag;     /* Zeiger auf geburtstag */
```

```
printf("%d", (*pgb).tag); /* tag ausgeben     */
(*pgb).tag = 17;          /* tag setzen     */
```

Zeiger auf Strukturen: Operator ->

- Kürzere Schreibweise mit dem *-> Operator:*

```
printf("%d", pgb-> tag);    /* tag auswaehlen    */
pmw -> tag = 17;            /* Feld setzen        */
```

- Der *-> Operator* dereferenziert und wählt ein Feld aus

```
(*pmw).tag
pmw -> tag
```

Ersetzt also '*' und '.' und behebt das Problem der sonst notwendigen Klammerung; meistens wird diese Schreibweise bevorzugt

Verkettete Strukturen

- Es gibt auch die Möglichkeit, dass ein Element einer Struktur ein Zeiger ist auf diese Struktur
- So kann man Strukturen miteinander verketten
- Dies wird im Zusammenhang zum Aufbau dynamischer Datenstrukturen, (z.B. verketteter Listen) verwendet

Verkettete (rekursive) Strukturen: Beispiel


```
typedef struct Datum {
    int tag;
    int monat;
    int jahr;
    struct Datum  *pnext;
} Datum;

Datum semStart, semEnd;
semStart.pnext = &semEnd;      /* Strukturen verketten */
```

make-Utility

Übersetzung auf der Kommandozeile

- Programme können auf der Kommandozeile übersetzt werden

```
c:\users\myproject>gcc hello.c -o hello.exe
```

- Aber oft nicht nur ein File sondern Dutzende/Hunderte/Tausende
- Linux Kernel 2011
  - 15 million lines of code
  - 33,000 files

https://arstechnica.com/information-technology/2012/04/linux-kernel-in-2011-15-million-total-lines-of-code-and-microsoft-is-a-top-contributor/

Das make-Utility

- Jedes etwas grössere Programm setzt sich aus einer Reihe von Dateien zusammen
- Schon bei Veränderungen einer Datei kann es nötig sein, mehrere andere Teile neu zu übersetzen
- Wird z.B. ein File verändert, so müssen alle andern die dieses File verwendet ebenfalls neu übersetzt werden
- Bei grossen Projekten kann das schnell sehr unübersichtlich werden
- Die Reihenfolge der Übersetzungen muss eingehalten werden
- Zusätzlich:
  - Compiler/Linker Optionen
  - Bei grossen Projekten: Es sollten jeweils nur die veränderten Teile neu übersetzt werden

beides durch gcc Programm

… Das make-Utility

- Das Utility make hilft, ein Programm aus mehreren Quellen unter Berücksichtigung der Abhängigkeiten zu erstellen
- Beim Aufruf arbeitet make die Regeln einer im aktuellen Verzeichnis liegenden **`makefile`**-Datei ab
- **`makefile`** das in dem Verzeichnis liegt, in dem make aufgerufen wird

```
c:\users\myproject>make
```

makefile und alle Quellen in demselben Verzeichnis

- Tutorials

  http://www.gnu.org/software/make/manual/make.html

  https://www.gnu.org/prep/standards/html_node/Standard-Targets.html

Wird ein C Programm als Quelle publiziert, dann i.d.R. mit einem dazugehörigen makefile

Makefile

- Eine Regel ist wie folgt aufgebaut:

```
#sample
target: dependencies
<TAB>   command
```

Einrückung mit tab zwingend
typische Fehlermeldung: "Missing Separator"

- **Target**: Was zu erstellen ist
- **Dependencies**: Legen fest, wann das zu geschehen hat, nämlich wenn eines von ihnen jünger ist als *target*
- **Command**: Das oder die abzuarbeitenden Kommandos;
  - Wichtig ist der Tabulator am Zeilenanfang
  - Befehle sind an die Linux Shell Befehle angelehnt (auch unter windows)
- **Kommentar:** wird ignoriert
- Da als *dependencies* auch andere *targets* angegeben werden können ist eine Verschachtelung der Regeln möglich

… Makefile – Beispiel

- Aufruf: `make` -> es wird erstes Target ausgeführt (z.B. all)
- Aufruf: `make build` -> Programm wird übersetzt
- Aufruf: `make run` -> Programm wird ausgeführt

```
# hello.exe wird uebersetzt und ausgeführt
all: clean build run
# loesche uebersetzte Dateien
clean:
   rm -f hello.exe
# Programm uebersetzen
build: hello.c
   gcc hello.c -o hello.exe
# Programm  starten
.SILENT: run #kein Echo des make run Kommandos
run:
   @echo == Run Program ==
   ./hello.exe
```

wenn hello.c geändert wurde, soll build ausgeführt werden

Ausgabe auf Konsole

"./" für aktuelles Verzeichnis

… Makefile – Beispiel Variablen

- Um Makefiles flexibel zu gestalten, führt man i.d.R. Variablen ein.

```
FILE=hello
all: clean build run
clean:
   rm -f $(FILE).exe
# Programm uebersetzen
build: $(FILE).c
   gcc $(FILE).c -o $(FILE).exe
# Programm  starten
.SILENT: run #kein Echo des make Kommandos
run:
   @echo == Run Program ==
   ./$(FILE).exe
```

… Makefile – Beispiel Compile/Link

outlook

- Das Programm ListTest wird aus den Files Lists.c und ListTest.c gebildet
- C Programme werden in einem ersten Schritt in Object-Files (.o) übersetzt
  - mit der Compiler Option -c
- Übersetzung in C eigentlich immer zweistufig:
  - **Compilation:** zuerst in sog. "o-Files"
  - **Linking:** "o-Files" in ausführbare Programme

```
# werden angegeben, um auch bei Veraenderungen daran
# die Übersetzung zu starten):
listtest: lists.o listtest.o
   gcc lists.o listtest.o -o listtest.exe
lists.o: lists.c lists.h
   gcc -c lists.c
listtest.o: listtest.c
   gcc -c listtest.c
```

Linken

Dependencies

-c nur Compilation ohne Linken

Generisches Makefile

outlook

- Mit Hilfe von Variablen lassen sich Regeln einfacher ändern (vor allem, wenn sie an mehreren Stellen benötigt werden)
- Beim Aufruf von *make* lassen sich auch Variablenzuweisungen übergeben – diese haben höhere Priorität als die Festlegungen im *Makefile*
- Man muss nicht alle Regeln und Variablen angeben – *make* verwendet dann seine Voreinstellungen
- Beispiel/Konventionen:
  - Variable `cc` für Compiler
  - Variable `cflags` für Compiler Flags

… Generisches Makefile: Automatic Variables

outlook

- $@ und $< sind sogenannte automatische Variablen
  - $< steht für das erste Prerequisite notwendig um das Output File zu erzeugen.
  - $@ steht für den Namen des Targets

- Beispiel:

```
hello.o: hello.c hello.h
         gcc -c $< -o $@
```

- Hier ist hello.o ist das Output File, zu dem $@ expandiert wird.
- Die erste Abhängigkeit ist hello.c, zu dem $< expandiert wird.

- Das -c Compiler Flag erzeugt das .o File.

… Generisches Makefile

outlook

```
# Variante mit Variablen, impliziten Regeln und
OBJS = lists.o listtest.o
BIN  = listtest.exe
CC = gcc
CFLAGS = -g

$(BIN): $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) -o $(BIN)

$(OBJS): %.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f *~ *.o $(BIN)
```

nur dieser Teil muss jeweils angepasst werden

die zu übersetzenden Quellen mit Endung .o

Namen des Programms

Target Rule: xyz.o depends on xyz.c,

The source file of the current dependency

prerequisite to buld target e.g. list.c

The target to be buld e.g lists.o

Noch Fragen?