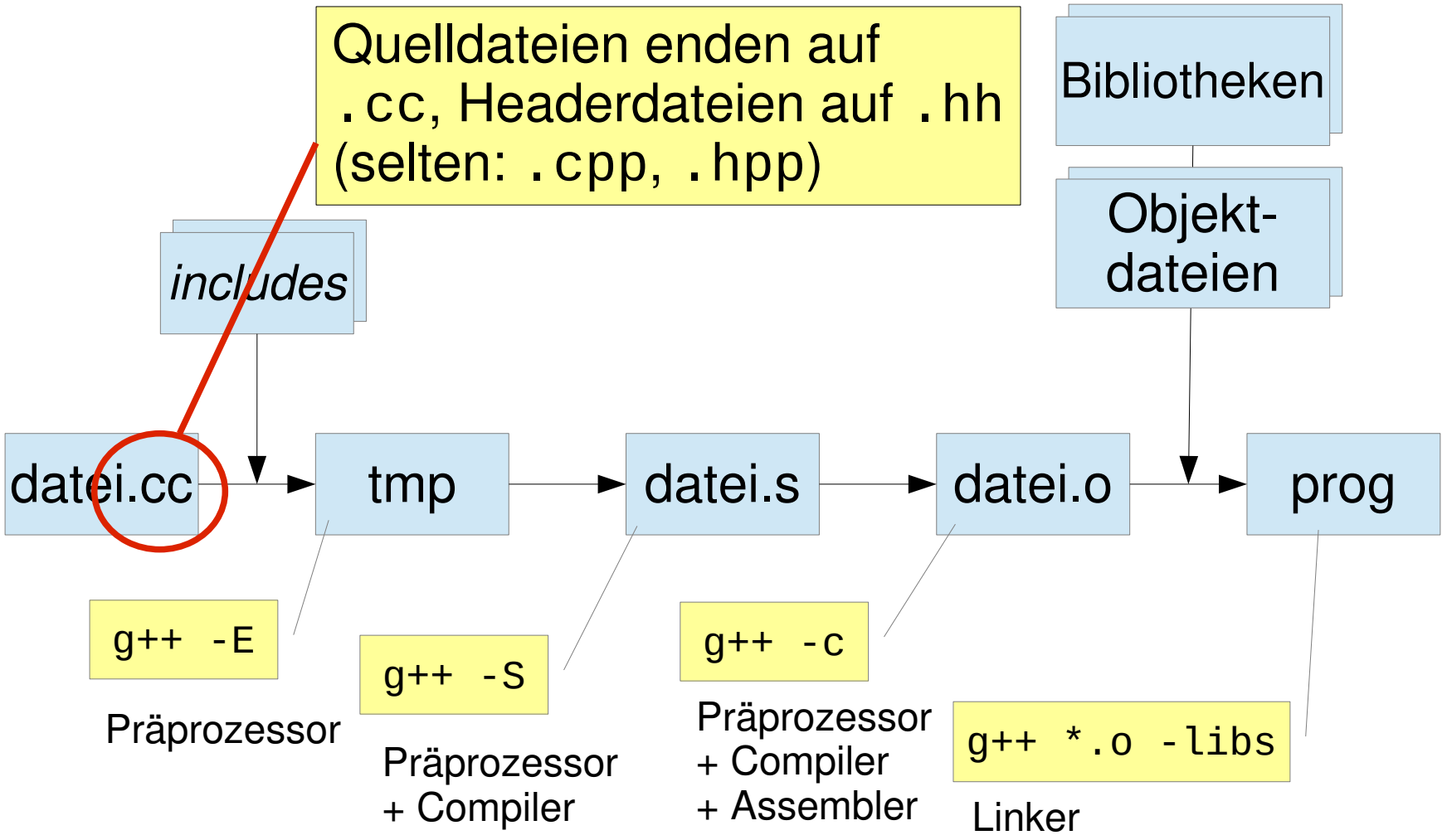
Modularisierung und getrennte Übersetzung

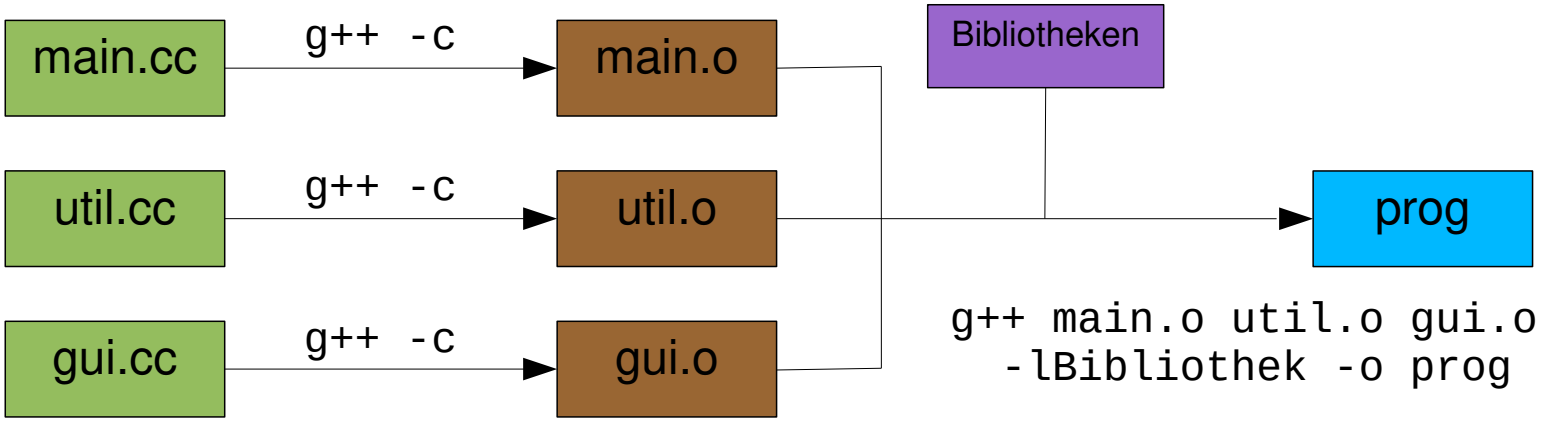
Vorkurs C/C++, Olaf Bergmann

Wdh.: Compilerpässe in C++

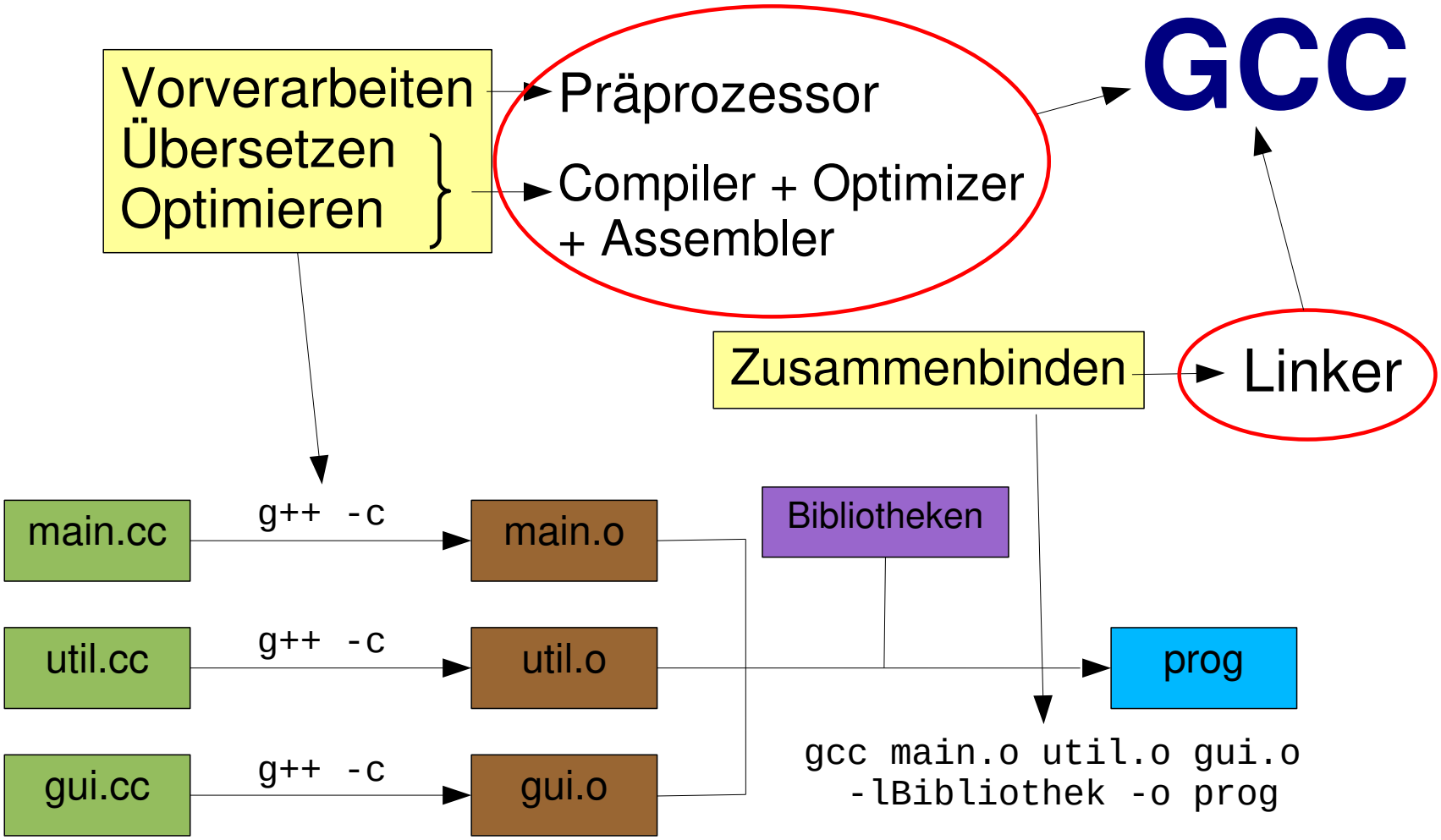


Modularisierung

- Größere Projekte auf mehrere Quelldateien aufteilen
  - Übersichtlicher
  - Ermöglicht paralleles Arbeiten
- Nur geänderte Teile neu übersetzen
- Linker fügt Teile zu ausführbarem Programm zusammen



Modularisierung



Der Linker

- `ld`
  - Eingabe: Objektdateien, Bibliotheken
  - Ausgabe: ausführbares Programm
  - Bindet Eingabedateien zusammen und löst Referenzen auf

`a.out, .exe`

- Objektdateien
  - plattformspezifischer Maschinencode
  - Nicht ausführbar, enthalten u. U. nicht aufgelöste Referenzen

Endung `.o`

- Bibliotheken
  - Statisch oder dynamisch („shared“)

Endung `.a, .so`

Symbole

- Alle verwendeten Symbole müssen aufgelöst werden
- Jedes Symbol darf nur einmal definiert sein

Beispiel: simple.cc

```
#include <gtkmm.h>

int main(int argc, char **argv) {
  auto app =
    Gtk::Application::create(argc,
     argv, "org.gtkmm.examples.base");

  Gtk::Window window;
  window.set_default_size(200, 200);

  return app->run(window);
}
```

https://developer.gnome.org/gtkmm-tutorial/stable/sec-basics-simple-example.html.en

Symbole

Wdh.

Makefile:

```
LINK.o=$(LINK.cc)
gtkmm_config = $(shell pkg-config gtkmm-$(GTK_VERSION) $1)

ifneq (${GTK_VERSION}, )
CPPFLAGS += $(call gtkmm_config,--cflags)
LDFLAGS += $(call gtkmm_config,--libs-only-L)
LDLIBS += $(call gtkmm_config,--libs-only-l)
endif
```

```
$ make example.o
c++ -pthread -I/usr/include/gtkmm-3.0 -I...
(→ example.o)

$ make example
c++ -o example example.o -lgtkmm-3.0 -latkmm-1.6 -lgdkmm...
(→ example)
```

Einbinden von `libgtkmm-3.0` usw.
→ sonst Fehler: ...undefined reference to '…'
collect2: error: ld returned …

Getrennte Übersetzung

```
int foo = 12;

int bar(double d) {
  ...
}
```

datei1.cc

```
extern int foo;
int bar(double d);

void baz() {
  foo = bar(1.0);
}
```

datei2.cc

- Einzelne Dateien übersetzen (`c++ -c ...`)
- Können Symbole aus anderen Dateien benutzen
  - Deklaration vor jeder Benutzung erforderlich
  - Alle Deklarationen müssen konsistent sein
  - Deklarationen müssen konsistent mit Definition sein
  - Jedes Objekt muss *genau einmal* definiert werden

Fehlerquellen

- Typische Fehler
  (1) mehr als eine Definition
  (2) Definition und Deklaration inkonsistent
  (3) keine Definition
  (4) Definition und Deklaration inkonsistent
- Fehlererkennung
  - Compiler sieht jeweils nur eine Quelldatei
  - (1) und (3) werden vom Linker erkannt
  - (2) und (4) nicht
    → Absturz zur Laufzeit möglich!

```
int foo = 12;

char b[10];

extern int c;

int f(int i) {...}
int g() {...}
```

datei1.cc

```
int foo;          (1)

extern char *b;   (2)

extern int c;     (3)

int f(double);    (4)
char *g();        (4)
```

datei2.cc

Headerdateien und der Präprozessor

Vorkurs C/C++, Olaf Bergmann

Header-Dateien

```
class Point {
  double x,y
public:
  void plot() const;
};

int f(int i);
```

myplot.hh

```
#include "myplot.hh"

void Point::plot() const {
  ...
}
```

myplot.cc

Konvention: eigene Header-Dateien enden auf .hh

- Ziel: Konsistente Deklarationen
  → Schnittstelle und Implementierung trennen

```
#include "myplot.hh"

int main() {
  Point().plot();
}
```

main.cc

Was gehört in Header-Dateien?

- Klassen
- Datendeklarationen
- Funktionsdeklarationen
- Typdefinitionen
- inline-Definitionen
- Aufzählungstypen
- Präprozessoranweisungen

```
class Point { … };
extern int v;
int f(double);
struct coord { int x, y };
inline int max(int a, int b) {...}
enum state { on, off };
#include …
```

Nicht hinein gehören

- Datendefinitionen
- Funktionsdefinitionen

```
int v;
int f() {...}
```

Präprozessoranweisungen (1)

`#include`

- Rein textuelle Ersetzung
- Compiler sieht nur fertige *Übersetzungseinheiten*

```
#include <iostream>
#include "mplot.hh"
```

`#define`

- Makro
- Textuelle Ersetzung, optional mit Parametern

```
#define PI 3.141
#define max(a,b) \
  ((a) > (b) ? (a) : (b))
```

`#undef`

- Präprozessprsymbole entfernen

```
#undef max
#undef HAVE_TIME_H
```

Einschub: Seiteneffekte von Makros

```
#define max(a,b) ((a) > (b) ? (a) : (b))

...
{
  int k, n = 13;
  k = max(++n, 4);   /* ((++n) > (4) ? (++n) : (4)) */
}
```

- Makro-Parameter werden textuell ersetzt
- Beispiel: mehrfaches Ausführen von `++n`
- Abhilfe: `inline`-Funktionen

```
inline int
max(int a, int b) {
  return a > b ? a : b;
}
```

Noch mehr Seiteneffekte von Makros

```
#define sq(x) x * x  /* falsch */

...
{
  double r;
  r = sq(y + 1);     /* == y + 1 * y + 1 */
}
```

- Makro-Parameter werden textuell ersetzt
- Beispiel: Prioritäten von Operatoren
- Abhilfe: Klammern (oder `inline`-Funktionen)

```
#define sq(x) ((x) * (x))
```

Präprozessoranweisungen (2)

`#if`, `#ifdef`, `#ifndef`

- Bedingter Abschnitt (bis `#endif`)
- `#ifdef X` ≡ `#if defined(X)`
- Optional `#else`, `#elif`

```
#if HAVE_SYS_TIME_H
r = gettimeofday(...);
#else
#error "no gettimeofday"
#endif
```

`#error`

- Beendet Übersetzung mit Fehler

`#pragma`

- Steuert Compiler-Spezifika

```
#pragma STDC FP_CONTRACT OFF
```

Benutzung von Header-Dateien

- Nach dem Ändern alle abhängigen Dateien neu übersetzen (→ make verwenden)
- „eigene“ Headerdatei einbinden
  - Compiler kann Deklarationen gegen Definitionen prüfen
- Geschachtelte `#includes` eher selten
  → Mehrfachdefinitionen möglich
  - Üblich: eine Headerdatei pro Quelldatei
  - Eventuall global Definitionen in eigener Headerdatei
    → `config.h`

Organisation von Headerdateien

```
#define LIBDIR "/usr/local/lib"
#define VERSION 3
#define HAVE_SYS_TIME_H 1
```

config.h

```
typedef struct buf_t {
  size_t size;
  char data[];
} buf_t;

buf_t *buf_create(size_t size);
void buf_delete(buf_t *);
```

buf.h

```
void error(char *);
```

error.h

```
#include "config.h"
#include "error.h"
#include "buf.h"

buf_t *
buf_create(size_t size) {
  ...
}
```

buf.c

```
#include <stdio.h>
#include "config.h"
#include "error.h"

void
error(char *msg) {
...
}
```

error.c

Zyklische Abhängigkeiten vermeiden

```
#ifndef MY_HEADER_H
#define MY_HEADER_H 1

#include "other_header.h"
...

#endif /* MY_HEADER_H */
```

my_header.h

Include-Guard: Schutz vor erneuter Ersetzung, nachdem MY_HEADER_H definiert ist.

```
#ifndef OTHER_HEADER_H
#define OTHER_HEADER_H 1

#include "my_header.h"
...

#endif /* OTHER_HEADER_H */
```

other_header.h

Include-Guard: Schutz vor erneuter Ersetzung, nachdem OTHER_HEADER_H definiert ist.

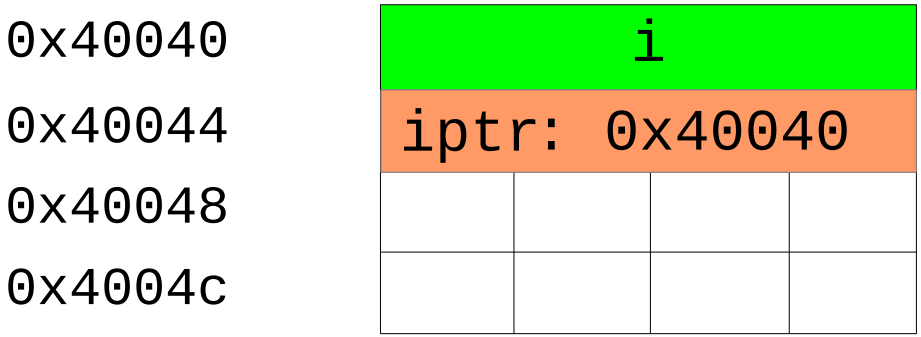
Pointer

Vorkurs C/C++, Olaf Bergmann

Pointer

- Wichtiges Konzept in C/C++
  - In Java versteckt → Umdenken erforderlich → häufige Fehlerquelle
- Pointer „zeigt“ auf ein Objekt im Speicher
  - Pointer-Variable enthält Speicheradresse, an der das Objekt liegt
  - Typ des Pointers impliziert Typ des Objekts
- Verwendet u. a. bei dynamischer Speicherverwaltung, direktem Hardwarezugriff
- Nicht unnötig einsetzen für Grunddatentypen und lokale Objekte

Pointer: Ein Beispiel (ILP32)




```
int i;
int *iptr;
```

iptr ist vom Typ „Pointer auf `int`“

Disclaimer:
mit C++ immer Smart Pointer benutzen!

```
iptr = &i;
```

Adressoperator

Operatoren

- Unärer Adressoperator &
  - Anwendbar auf Variablen und Arrayelemente
  - Nicht auf Werte, Registervariablen

```
int i, a[10];
unsigned u;
int *ptr;

ptr = &i;
ptr = &a[10];

ptr = &u;
ptr = &3;
ptr = &(i+3);
```

Fehler!

- Dereferenzieren: unärer Operator *
  - Umkehrung von &
  - Liefert Objekt, auf das ein Pointer zeigt

```
int i, j = 5;
int *ptr = &i;

*ptr = 1; /* i == 1*/
ptr = &j;
i = *ptr; /*i == 5*/

i = ptr;
```

Fehler!

Pointer-Beispiele

```
int i, j = 5;
int *ptr;         /* "*ptr ist ein int"  */
char *p, *q;      /* "*p und *q sind char"  */
double *dp, d;    /* d ist kein Pointer! */

ptr = &i;

*ptr = 1;         /* i ist jetzt 1 */
ptr = &j;

i = *ptr;         /* i ist jetzt 5 */
*ptr += 2;        /* j ist jetzt 7 */
++*ptr;           /* j ist jetzt 8 */
(*ptr)++;         /* j ist jetzt 9 */

i = ptr;          /* falscher Typ */
dp = &i;          /* falscher Typ */
```

Fehler!

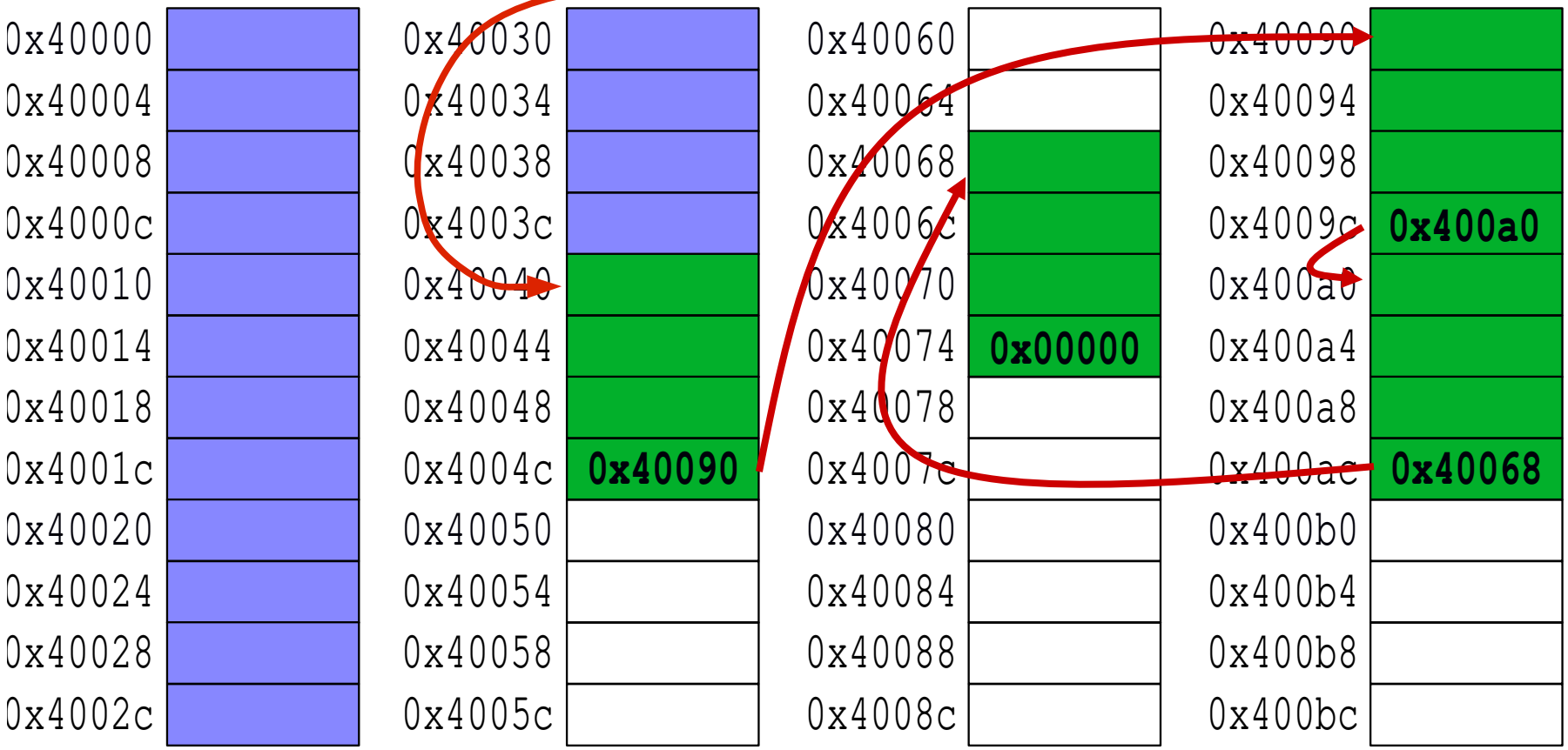
Beispiel: Listenelement in C

```
struct verpackung {
    int laenge;
    int breite;
    int hoehe;
    struct verpackung *next;
};
```

Disclaimer:
C++ kann's besser

Einfach verkettete Liste

```
struct verpackung *first;
```



Pointer und Funktionsargumente

- Argumentübergabe in C ist call-by-value
- Call-by-reference über Pointer (in Java automatisch)

```
void swap(int *p, int *q) {
  int temp = *p;
  *p = *q;
  *q = temp;
}
```

- Benutzung mit Adressoperator

```
int x, y, a[10];
swap(&x, &y);
swap(&a[0], &a[1]);
```

Ergebnisparameter

```
#include <ctype.h>
#include <stdio.h>
```

```
int getint(int *result) {
  int c, ok = 0;

  *result = 0;
  c = getc(stdin);
  if (isdigit(c)) {
    do {
      *result = 10 * *result + (c – '0');
      c = getc(stdin);
    } while(c != EOF && isdigit(c));
    ok = 1;
  }

  if (c != EOF) {
    ungetc(c, stdin);
  }

  return ok;
}
```

Disclaimer:
mit C++ immer Referenzen benutzen!

```
        ...
int i;

if (getint(&i)) {
  /* i is valid... */
} else {
  /* error... */
}
        ...
```

NULL und nullptr

```
#include <stddef.h>

char *p = NULL;

if (p != 0) ≡ if (p)
```

- Spezieller Wert für Pointer
  - Zeigt auf „nichts“
  - Darf nicht dereferenziert werden
- Implizite Typumwandlung `0` → `NULL`
  - Integer `0` darf an Pointer zugewiesen werden
  - Pointer kann mit `0` verglichen werden
  - `(Typ *)0` kann nie valide Adresse sein
- Implizite Typumwandlung in booleschen Ausdrücken, automatische Umtypung

Disclaimer: mit C++ immer `nullptr` benutzen!

void *

- Generischer Pointertyp
  - Cast von Pointer nach `void *` und zurück ist erlaubt
    Typ * → void * → Typ * ist invariant
- `void *` kann nicht dereferenziert werden

```
int i;
int *ip = &i;  /* Pointer auf integer */
char *cp;      /* Pointer auf character */
void *vp;

vp = ip;       /* erlaubt */
vp = cp;       /* erlaubt */

ip = vp;       /* erlaubt (nicht in C++) */
```

Arrays

Vorkurs C/C++, Olaf Bergmann

Pointer und Arrays

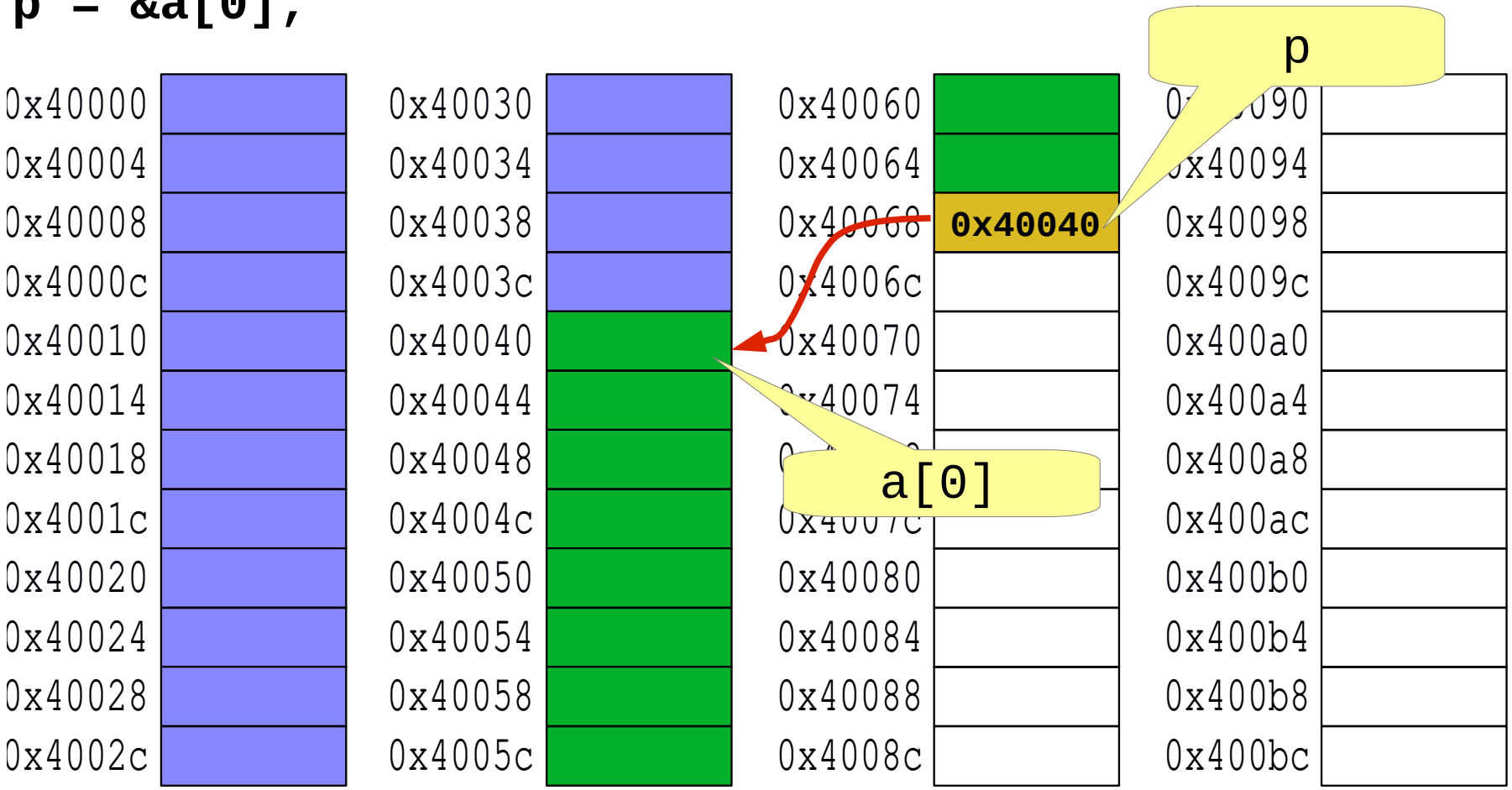
- Arrays im Prinzip aus Java bekannt
- Enger Zusammenhang zwischen Pointer und Array
  - Pointer-Zugriff statt Array-Indizierung möglich
  - Gerade zu Beginn häufige Fehlerquelle
- Pointer können auf Array-Elemente zeigen
  → Pointer enthält Adresse von Array-Element

```
int a[10], *p;
p = &a[0];          /* zeigt auf erstes Array-Element
*/
```

- Dereferenzierung liefert Inhalt von Array-Element: `*p` ≡ `a[0]`

Pointer auf Array-Element

```
int a[10], *p;
p = &a[0];
```



Pointer-Arithmetik

- `p+1` nächstes Array-Element
- `p+i` i-tes Element hinter p
- `p-i` i-tes Element vor p

`i` ist typabhängig:
Anzahl der Bytes, auf die der Pointer zeigt

```
int a[10], *p;
p = &a[0];

*p              /* liefert a[0] */
*(p+1)          /* liefert a[1] */
*(p+i)          /* liefert a[i] */

p += 5;
*p              /* liefert a[5] */
*(p-1)          /* liefert a[4] */
*(p-i)          /* liefert a[5-i] */
```

Äquivalente Schreibweisen

- Array ist identisch mit Adresse des ersten Elements
  `p = &a[0]` ≡ `p = a`
- Array/Index-Ausdruck entspricht Pointer/Offset-Ausdruck

Äquivalente Schreibweisen:

`a[i]`	identisch mit	`*(a+i)`
`&a[i]`	identisch mit	`a+i`
`&a[0]`	identisch mit	`a`

Pointer indizieren:

`p[i]`	identisch mit	`*(p+i)`
`p[-1]`	identisch mit	`*(p-1)`

Unterschiede

- Compiler verwandelt Array-Name in Pointer auf erstes Element
  - Namen von Array-Variablen sind konstant
  - Kein zusätzlicher Speicher-platz für Pointervariable benötigt

```
int a[10], *p;

a++;            /* Fehler! */
a = p;          /* Fehler! */
```

Achtung!

`sizeof(a)` → 40

`sizeof(p)` → 4 bzw. 8

`sizeof(a)/sizeof(int)` ≡ Anzahl Elemente in a

Funktionsargumente

- Umwandlung Array ↔ Pointer
- Beispiel: Länge eines Strings (char-Array):

```
int strlen(const char *s) {
  int n;
  for (n = 0; *s != '\0'; s++, n++);
  return n;
}
```

- Funktion arbeitet auf lokaler Kopie des Pointers
- Aufruf mit Array oder Pointer

```
strlen("Hi there!");        /* Konstante */
strlen(array);              /* char array[80]; */
strlen(ptr);                /* char *ptr; */
```

Deklaration des formalen Parameters

```
int strlen(const char s[]);
                oder
int strlen(const char *s);
```

- Argument wird immer in Pointer umgewandelt
  ⇒ Deklaration als Pointer klarer

Pointer-Vergleich

```
int size = 7;
int a[size], *p;

for (p = a; p < a+size; p++) {
  *p = 0;
}
```

- Pointer können verglichen werden (<, >, ==, !=)
- Beide Pointer sollten(!) in dasselbe Array zeigen
- Sonderfall: Pointer direkt hinter Array OK (`&a[size]`)
- == und != immer definiert

Pointer-Differenz

- Pointer dürfen subtrahiert werden (nicht addiert!)
- Ergebnistyp: `ptrdiff_t` (integer-Typ)
  - Anzahl der Elemente zwischen den Pointern

Beispiel: vereinfachtes strlen:

```
int strlen(const char *s) {
  const char *p = s;
  while (*p != '\0')
    p++;
  return p - s;
}
```

Character-Pointer und Strings

- String-Konstante ist ein Character-Array
- Automatisch mit '\0' terminiert („Null-Byte“)
- Bei Funktionsaufruf:

```
speak("Hello, world");
```

  - Funktion erhält Pointer als Argument
- Bei Zuweisung:

```
char *msg;
msg = "this is a string";
```

Pointer auf erstes Zeichen
String wird nicht kopiert!

Zeichenketten kopieren

- Entwickler ist für Speichermanagement verantwortlich
  Beispiel: Kopieren einer Zeichenkette

```
char msg[] = "hello";
char buf1[6]; /* 5 chars + Null-byte */
strncpy(buf1, msg, strlen(msg) + 1);
```

Disclaimer:
mit C++ immer Strings und Smart Pointer benutzen!

Dynamischen Speicher reservieren ...

```
char *buf2 = malloc(sizeof(msg));
if (buf2) {
  strncpy(buf2, msg, 6);
  ...
}
free(buf2);
```

... und wieder freigeben

besser:

```
char *buf3 = strdup(msg);
...
free(buf3);
```

Array und Pointer bei Deklaration

```
const char amsg[] = "hello";
const char *pmsg  = "hello";
```

- `amsg` ist konstant, `psmg` änderbar
- `amsg` enthält Speicherplatz für 6 chars
- `pmsg` enthält Speicherplatz für Adresse
- Erste Definition: 6 Bytes (Beispiel)
- Zweite Definition: 4/8 Bytes + 6 Bytes (Beispiel)

Character-Pointer und Funktionen

- C kennt keine Operatoren für Stringverarbeitung
  → Bibliotheksfunktionen (string.h)
- Beispiel: `strcpy(s, t)` kopiert String `t` nach `s`
- Array-Version von strcpy:

```
void strcpy(char *s, char *t) {
  int i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

Häufiges Idiom:

```
while (*s++ = *t++);
```

Pointer-Arrays, Pointer auf Pointer

- Pointer-Array-Deklaration
  `char *names[12];`
- Definition mit Initialisierung (für globale Variablen)
  `char *names[] = { "Januar", "Februar", ... };`
- Pointer-Array degeneriert zu Pointer auf Pointer in Ausdrücken: `char **ptr = names;`

Als Schleife:

```
int i;
for (i = 0; i < 12; i++)
  printf("%s\n", names[i]);
```

Mit Pointern:

```
char **p;
for (p = names; p < names+12; p++)
  printf("%s\n", *p);
```

Argumente der Funktion main()

```
int main(int argc, char **argv)
```

- `argv`: Array von Strings
- `argc`: Anzahl der Elemente in argv
- `argv[0]`: Programmname
- Rückgabewert: Fehlercode
  z. B. `echo $?`

Funktionspointer

- Deklaration

```
int (*p)(int);          char *(*q)();
```

- Zuweisung und Aufruf

```
int add(int a, int b) {
  return a + b;
}

int i, (*f)(int, int);
f = &add;
i = (*f)(1, 2);
void (*strfun)(char *, char *);
strfun = &strcpy;
```

& und * können entfallen:

```
f = add     ≡  f = &add
f(1,2)      ≡  (*f)(1,2)
```

Typdefinitionen

- Schlüsselwort `typedef`

```
typedef int size_t;
typedef struct point {
  double x;
  double y;
} point;
```

- typedef und Funktionspointer

```
typedef void (*strfunc)(char *, char *);
strfunc strfun = strcpy;
```

Mehrdimensionale Arrays initialisieren

```
int matrix[2][5] = {
   { 2, 5, 0, 0, 1 },
   { 0, 9, 1, 2, 4 }
};
```

Erste Dimension darf implizit sein

```
char *days[][7] = {
   { "Monday", "Tuesday", ... },
   { "Montag", "Dienstag", ... }
};
```

Array-Loop mit Pointer

```
struct key *p;

for (p = keytab; p < keytab + nkeys; p++) {
  if (p->count > 0) {
    printf("%d %s\n", p->count, p->word);
  }
}
```

Tabellengröße berechnen

- automatisch

```
int nkeys = sizeof(keytab) / sizeof(key);
int nkeys = sizeof keytab / sizeof *keytab;
```

- Zwei Formen von sizeof:

`sizeof` *Objekt* `sizeof(`*Typname*`)`