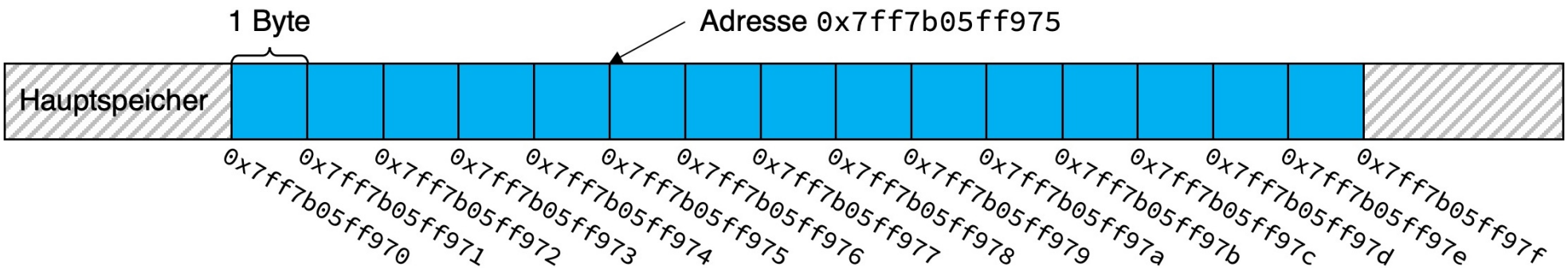
4. GRUNDLAGEN II

Inhalt

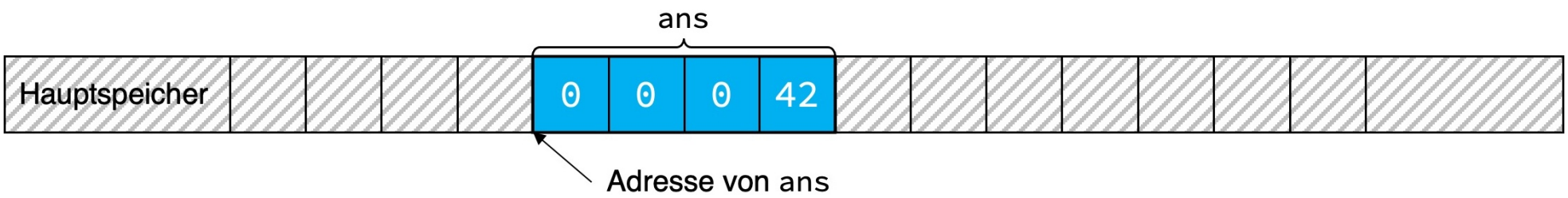
- Felder, Zeiger und Referenzen
- Benutzer:innendefinierte Typen
  - Strukturen
  - Klassen
  - Enumerationen
  - Unions
- Debugging mit `gdb`

Hauptspeicher



- Konzeptionell ist der **Hauptspeicher** eines Rechners eine Sequenz von Speicherzellen
  - Jede Speicherzelle ist **1 Byte** $=$ **8 Bits** groß, fasst also eine Zahl von `0` bis und mit `255`
  - Die Speicherzellen sind fortlaufend nummeriert
  - Die Nummer einer Speicherzelle heißt **Adresse**

Variablen

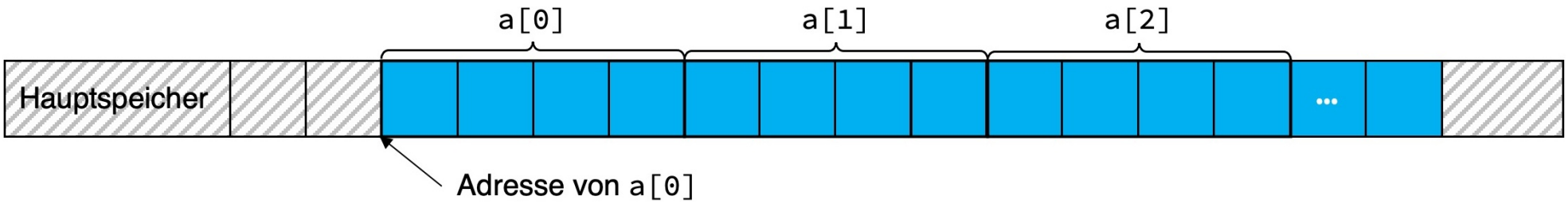


- Wir wissen schon: Eine Variable ist eine Abstraktion für ein Stück Speicher

```
int ans = 42;    // int-Zahl (typischerweise 4 Bytes)
```

  - Je nach Typ belegt die Variable ein oder mehrere Bytes
  - Die Anzahl Bytes kann mit `sizeof` abgefragt werden
- Der Variablenname steht für den **Wert** (interpretiert gemäß des Typs der Variablen) in den belegten Speicherzellen
- Die **Adresse** der Variable im Speicher erhält man mit dem Operator `&` (mehr dazu später)

Felder



- **Felder** („arrays“) sind Folgen von Variablen
  - Alle Elemente haben den gleichen Namen und den gleichen Typ
  - Zugriff auf Elemente erfolgt mit dem `[]` Operator über ihren Index
  - Die Elemente stehen hintereinander im Speicher
- Der Feldname `a` steht für die Adresse (nicht den Wert) des ersten Elements
  - Richtige Informatiker:innen beginnen bei `0` zu zählen!
  - Auf den Wert des ersten Elements wird mit `a[0]` zugegriffen
- **GTK** Die Syntax `a[i]` ist lediglich eine Abstraktion für die Adressberechnung: `a` + `i` × `sizeof(`Elementyp von `a)`

Felder

- Wie bei „normalen“ Variablen kann man den Feldelementen schon bei der Deklaration Werte zuweisen

```
int a[3];           // Uninitialisiertes Feld mit 3 Elementen
int b[3] = { 1, 2, 3 };
int c[3] = { 1, 2, 3, 4 };  // Compilefehler: zu viele Werte
int d[3] = { 1, 2 };              // a[2] mit 0 initialisiert
int e[3] = { 0 };       // Alle Elemente mit 0 initialisiert
```

- **GTK** Ohne Initialisierung ist der Inhalt der betreffenden Speicherzellen gemäß C/C++-Standard beliebig
  - Manche Systeme initialisieren die Felder trotzdem mit Null
  - Darauf sollte man sich aber nicht verlassen!
- **GTK** Felder spielen in der C++-Programmierung eine ähnliche Rolle wie der `vector`, den wir letzte Woche kennengelernt haben
  - Felder sind aber auf einer **tieferen** Abstraktionsebene
  - Das heißt man muss mehr selber machen
  - Wir verwenden `vector`, wenn immer möglich

Mehrdimensionale Felder

- Ein **zweidimensionales** Feld deklariert man z.B. so

```
int m[2][4];   // Speicher fuer 2 * 4 ints
```

- Das Feld `m` kann wie folgt initialisiert werden

```
int m[2][4] = { { 0, 1, 2, 3 }, { 4, 5, 6, 7 } };
```

- Unser Compiler akzeptiert auch

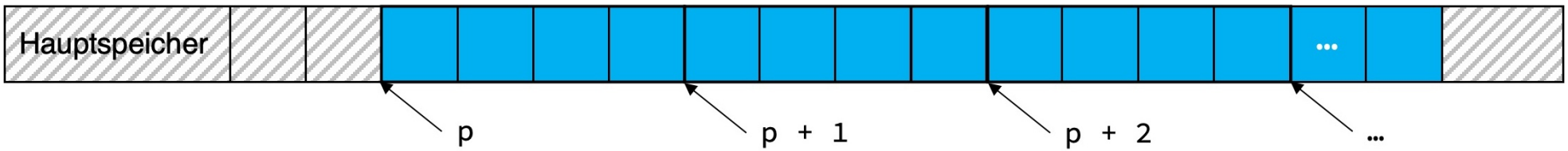
```
int m[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
```

  - Das ist aber **kein** guter Stil!

- Mehrdimensionale Felder sind aber **nicht** äquivalent zu eindimensionalen Feldern

```
cout << m[1][3]; // Gibt das Element 1 * 4 + 3 = 7 aus
cout << m[7];    // Compiliert nicht!
```

Zeiger



- Zeiger („pointers“) sind Variablen, deren Wert eine **Adresse** ist
  - Bei der Deklaration gibt man an, wie der Inhalt des Speichers an dieser Adresse zu interpretieren ist

```
int* p;        // Ein Zeiger auf einen int
```

  - Zugriff auf den Wert (**dereferenzieren**) mit dem Operator `*` vor dem Variablennamen

```
cout << *p;  // Gibt den Wert aus, auf den p zeigt
```

- GTK Diese Zeiger werden häufig als **Raw Pointers** bezeichnet
  - Damit soll der Gegensatz zu den **Smart Pointers**, über die wir letzte Woche geredet haben, zum Ausdruck gebracht werden
  - Raw Pointers sind auf einer **tieferen** Abstraktionsebene

Zeiger

- **Achtung**: Das Sternchen („asterisk“) hat in C++ (im Zusammenhang mit Zeigern) zwei **verschiedene** Funktionen
  - In der Variablendeklaration drückt das Sternchen **zwischen** Typnamen und Variablennamen aus, dass es sich um einen Zeiger handelt

```
int* p;       // Ein Zeiger auf einen int
```

  - In einem Ausdruck drückt der Operator `*` **vor** dem Variablennamen aus, dass auf den Wert und nicht auf die Adresse zugegriffen wirdl

```
cout << *p;  // Gibt den Wert aus, auf den p zeigt
```

- GTK Die C++-Syntax schreibt nicht vor, dass das Sternchen **direkt** am Typ- oder Variablennamen dran stehen muss.

```
int* p1;    // Ein Zeiger auf einen int
int * p2;   // Auch ein Zeiger auf einen int
int *p3;    // Nochmals ein Zeiger auf einen int
```

Coding-Konventionen halten den Code einheitlich und leserlich!

Null-Zeiger

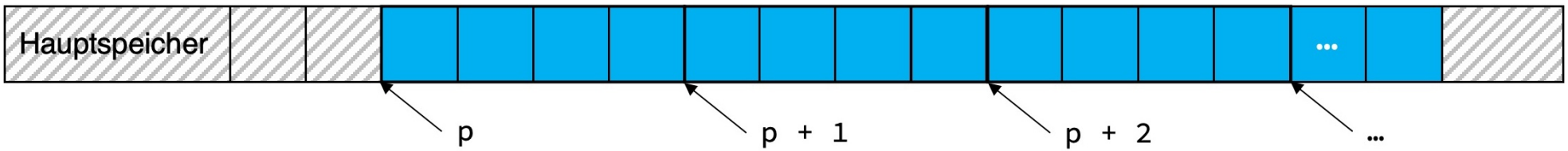
- **Achtung**: Wenn ein uninitialisierter Zeiger dereferenziert wird, führt das fast immer dazu, dass das Programm abstürzt

```
int* p;       // Ein uninitialisierter Zeiger
cout << *p;   // Schutzverletzung!
```

- Es ist also noch eine ganz schlaue Idee, Zeiger zu initialisieren
  - mit einer Adresse eines Speicherbereichs, der dem Programm gehört
  - mit dem Null-Zeiger `nullptr`
- Den Null-Zeiger darf man aber auch **nicht** dereferenzieren!
- Man kann aber einfach prüfen, ob ein Zeiger der Null-Zeiger ist

```
if (p) {
  int a = *p;  // Hier kann nichts mehr schiefgehen!
}
```

Zeigerarithmetik



- Zeigerarithmetik bedeutet, dass man mit Zeigern rechnen kann!
  - Zu einem Zeiger kann man zum Beispiel eine Zahl addieren
  - Diese wird dann automatisch mit der Größe des Typs multipliziert

```
cout << *(p + 2);   // Gibt den int-Wert an der Adresse
                    // p + 2 * sizeof(int) aus
p = p + 3;          // Inkrementiert die Adresse um
                    // 3 * sizeof(int)
```

- GTK Zeigerarithmetik kann grausam in die Hose gehen!
- GTK Smart Pointers können keine Zeigerarithmetik

Zeiger vs. Felder

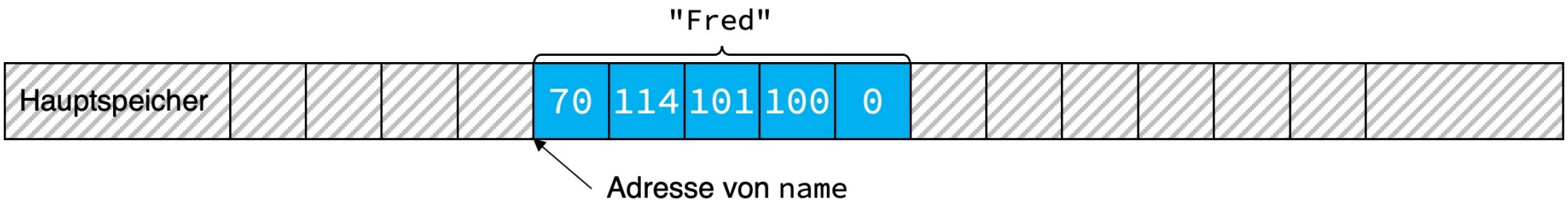
- Den **Namen** eines eindimensionalen Feldes kann man benutzen wie einen Zeiger auf das erste Element des Feldes

```
int fib[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int* p = fib;
cout << *p;  // Gibt 0 aus
```

- Der Zugriff über den Operator `[]` macht **exakt** das gleiche wie die Zeigerarithmetik auf der vorherigen Folie

```
int fib[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int* p = fib;
cout << p[3];       // Gibt 2 aus
cout << *(p + 3);  // Macht exakt das gleiche!
```

Zeichenketten (C-Strings)



- Wir haben in der zweiten Woche schon gesehen, dass es in C++ (mindestens) zwei Arten von Zeichenketten („Strings“) gibt
  - Bisher haben wir, wenn immer möglich, mit `std:string` gearbeitet
  - Heute wollen wir nun auch die sogenannten **C-Strings** verstehen
- Ein C-String ist ein Feld bzw. ein Zeiger von Elementen des Typs `char` (ein Zeichen)

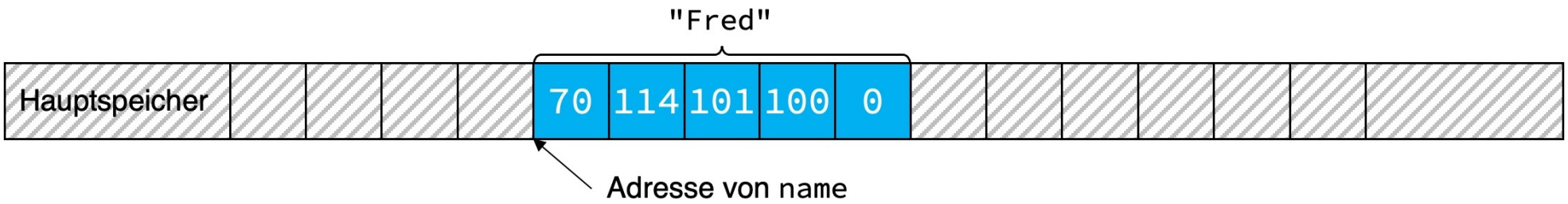
```
char name[5] = { 'F', 'r', 'e', 'd', 0 }
char* p = a + 3;  // Zeigt auf die Adresse von 'd'
```

- C-Strings kann man aber auch einfacher initialisieren

```
const char* name = "Fred";  // s zeigt auf das 'F'
```

  - **GTK** Ohne `const` warnt der Compiler, da `"Fred"` eine Konstante ist

Zeichenketten (C-Strings)



- Strings in C/C++ sind **nullterminiert** („null terminated“)
  - Die Länge des Feldes ist immer größer als die Länge des Strings
  - Das Zeichen `0` zeigt an, wo der String aufhört
  - **Beispiel**: Für `"Fred"` wird Platz für **fünf** Zeichen reserviert, da am Ende noch das Zeichen `0` steht
- Über `<cstrings>` können nützliche Funktionen eingebunden werden, um mit Strings dieser Art zu arbeiten
  - `strcat` verkette zwei Strings
  - `strcpy` kopiert einen String in einen anderen
  - `strlen` gibt die Länge des Strings (nicht des Feldes) zurück

Zeiger auf Zeiger

- Zeiger können auf **alles** zeigen, auch auf andere Zeiger
- Ein typisches Beispiel sind Zeiger auf Zeichenketten
  - Zeichenketten sind vom Typ `char*` (Zeiger auf `char`) sind
  - Ein Zeiger auf eine Zeichenkette ist also vom Typ `char**`

```
char* s[3] = { "ene", "mene", "muh" };       // Elementtyp char*
cout << s[1];                                 // Gibt "mene" aus

char** p = s;                        // Verwende Feld X wie Zeiger auf X
cout << *(p + 1);                             // Gibt auch "mene" aus
```

- Nun verstehen wir auch die Signatur der `main`-Funktion völlig
  - `argc` ist ein ganze Zahl, die die Anzahl der Argumente angibt
  - `argv` ist ein Feld von C-Strings, das die Argumentwerte beinhaltet
- **GTK** Da `char* argv[]` und `char** argv` äquivalent sind, sieht man auch häufig die zweite Variante in der `main`-Signatur

Adress-Operator

- Der Adress-Operator `&` („address-of operator“) gibt die Speicheradresse einer Variable zurück

```
int x = 27;    // Der Wert von x ist 27
int* p = &x;   // Der Wert von p ist die Adresse von x
int* q = &x;   // p und q zeigen an die gleiche Adresse
```

- **GTK** Der Adress-Operator `&` ist das **direkte** Gegenstück zum Dereferenz-Operator `*` („dereference operator“)
- Wie das Sternchen hat auch das Kaufmannsund („ampersand“) noch eine zweite Funktion in diesem Zusammenhang

```
int x = 27;    // Der Wert von x ist 27
int& y = x;    // y ist ein anderer Name (Alias) fuer x
y = 5;
cout << x;     // Gibt 5 aus
```

- **GTK** Eine mit & deklarierte Variable heißt **Referenz**

Zeiger vs. Referenz

Eine Referenz kann man sich auch als Zeiger vorstellen, der immer **automatisch** dereferenziert wird.

- Mit einem **Zeiger**

```
int i = 21;
int* p = &i;
*p = 42;
cout << *p;
```

- Mit einer **Referenz**

```
int i = 21;
int& a = i;
a = 42;
cout << a;
```

- Im Gegensatz zu einem Zeiger kann man bei einer Referenz **nicht** verändern, worauf sie zeigt
- Referenzen werden vor allem bei der **Übergabe von Argumenten** an oder aus einer Funktion benenutzt, z.B.
  - weil der Wert in der aufgerufenen Funktion verändert werden soll
  - weil es zu teuer wäre, den ganzen Wert in die Funktion zu kopieren

Das ganze Leben ist ein Quiz

- Manchmal ist die Syntax für Zeiger auf Felder etwas konfus, weil man `*` vor die Variable schreibt und `[]` danach

```
int a[3] = { 1, 2, 3 };           // Feld von drei ints
int (*p)[3];           // Zeiger auf Feld von drei ints
p = &a;                                 // p zeigt nun auf a
int* q[3];                  // Feld von drei int-Zeigern
```

- Für zwei- oder mehrdimensionale Felder funktioniert es analog

```
int m[3][3] = { ... };  // Feld von drei mal drei ints
int (*p)[3][3];            // Zeiger auf ein solches Feld
p = &m;                                 // p zeigt nun auf m
(*p)[3][2];  // Elementzugriff im Feld auf das p zeigt
```

- C-gibberish↔English: `https://cdecl.org/`

Dynamische Speicherverwaltung

- Im letzten Kapitel haben wir im Zusammenhang mit Smart Pointers bereits über **zwei Arten** der Speicherverwaltung geredet
  - **Statische Speicherverwaltung** auf dem Stack
  - **Dynamische Speicherverwaltung** auf dem Heap
- In diesem Kapitel haben wir allen Speicher **statisch** reserviert

```
int x;              // Reserviert  4 Bytes auf dem Stack
int a[5];           // Reserviert 20 Bytes auf dem Stack
```

- Statische Speicherverwaltung hat aber ihre Grenzen

```
int n = atoi(argv[1]);   // 1. Kommandozeilenargument
int b[n];   // Compiliert nicht: n muss konstant sein
```

- Mit `new` kann man zur Laufzeit Speicher **dynamisch** reservieren

```
int* b = new int[n];    // Zeiger auf n ints
```

Dynamische Speicherveraltung

- Dynamisch reservierter Speicher muss vom Programm **explizit** wieder freigegeben werden
  - `delete` gibt ein einzelnes Objekt frei
  - `delete[]` gibt ein Feld von egal was frei
- Mit `new` und `delete` können schnell Fehler passieren
  - **Schutzverletzung** („sementation fault“): Zugriff auf Speicher, der noch nicht reserviert oder bereits wieder frei gegeben wurde
  - **Speicherleck** („memory leak“): Speicher, auf den man keinen Zugriff mehr hat, wurde nicht freigegeben
- Solche Fehler sind typischerweise übelst knifflig zu finden
  - Oft äußern sich diese Fehler nicht an der Stelle, wo sie passieren, sondern erst (viel) später im Programmablauf
  - Das Tool `valgrind` erleichtert die Suche nach solchen Fehlern

Dynamische Speicherverwaltung

- Benutzung von `valgrind`
  - Ausführbares Programm einfach als Argument übergeben

```
> valgrind ./list
```

  - `valgrind` findet vor allem Zugriffsfehler auf dynamisch reservierten Speicher auf dem Heap
  - Insbesondere das `LEAK SUMMARY` ist hier informativ

```
definitely lost: 16 bytes in 1 blocks
indirectly lost: 112 bytes in 7 blocks
```

  - Noch mehr Details erhält man mit

```
> valgrind -leak-check=full ./DoublyLinkedList
```

- **GTK** Wird das Programm mit der Option `-g` (wie beim Debuggen) kompiliert, erhält man Hinweise mit Zeilennummern!

Benutzer:innendefinierte Typen

- In der zweiten Woche haben wir bereits über die Komponenten des **Typsystems** einer Programmiersprache gesproche
  - **Basistypen** („built-in types“)
  - Konstruktoren für **zusammengesetzte Typen**
  - Regeln zur **Typisierung** von Ausdrücken
- Mit `[]` (Felder), `*` (Zeiger) und `&` (Referenzen) haben wir schon drei **Konstruktoren** für zusammengesetzte Typen gesehen
  - Typkonstruktoren funktionieren nach dem „LEGO-Prinzip“
  - Sie bauen aus bestehenden Typen neue, komplexere Typen
- Im Folgenden schauen wir uns noch vier weitere Konstruktoren für zusammengesetzte Typen an
  - `enum` konstruiert **Aufzählungen** („enumerations“)
  - `struct` konstruiert **Strukturen** („structures“)
  - `class` konstruiert **Klassen** („classes“)
  - `union` konstruiert **Unions** („unions“)

Aufzählungen

- Ein **Aufzählungstyp** definiert alle Werte seines Wertebereichs bei der Deklaration mit einem eindeutigen Namen
  - Aufzählungen verwendet man oft anstatt (numerischer) Konstanten
  - Dadurch wird der Code lesbarer als mit den nackten Zahlenwerten
- In C++ gibt es zwei Arten von Aufzählungstypen
  - **untypisierte** Aufzählungen (`enum`)
  - **typisierte** Aufzählungen (`enum class`)
- Wir verwenden, wenn immer möglich, **typisierte** Aufzählungen

```
enum class Farbe { rot, gruen, blau };
enum class Ampel { rot, gelb, gruen };
```

- Typisierte Aufzählungen sind **zuweisungssicher**

```
Farbe a = rot;          // Fehler: welches rot?
Farbe b = Ampel::rot;   // Fehler: Ampel::rot ist keine Farbe
Farbe c = Farbe::rot;   // Passt
auto d = Farbe::rot;    // Passt: Farbe::rot ist eine Farbe
```

Strukturen

- Eine **Struktur** ist ein Verbundtyp („composite type“)
  - Sie gruppiert verschiedene Datenelemente unter einem Namen
  - Die Datenelemente heißen Mitglieder („members“)
  - Die Mitglieder einer Struktur können unterschiedliche Typen haben
- In C++ können auch Funktionen Mitglieder einer Struktur sein
- Die Mitglieder einer Struktur werden mit dem Mitglieds-zugriff-Operator `.` („member access operator“) zugegriffen

```
struct Film {
  string titel;
  int dauer;
  int jahr;
};

Film f;
f.titel = "Ey_Mann,_wo_is'_mein_Auto?";
```

Klassen

- Eine **Klasse** ist ebenfalls ein Verbundtyp
  - „Klasse“ ist ein Begriff der objektorientierte Programmierung
  - Dort beschreibt sie eine Menge gleichartiger Objekte
- GTK Bis auf ein Detail gibt es in C++ keinen **fundamentalen** Unterschied zwischen Strukturen und Klassen
- Wir kommen in Kapitel 6 und 7 darauf zurück, wenn wir über objektorientierte Programmierung sprechen

Unions

- Eine **Union** ist auch ein Verbundtyp
  - Im Gegensatz zu Strukturen und Klassen beginnen aber alle Mitglieder einer Union an der **gleichen** Speicheradresse
  - Man kann also immer nur **einen** Wert in einer Union speichern
  - Unions sind in der Praxis eher selten

```
union EntwederOder {
  int i;
  char c;
};

union EntwederOder x;
x.i = 65;
cout << x.i << ", " << x.c << endl; // Ausgabe: 65, A
x.c = 'B';
cout << x.i << ", " << x.c << endl; // Ausgabe: 66, B
```

Mitgliedszugriff-Operator für Zeiger

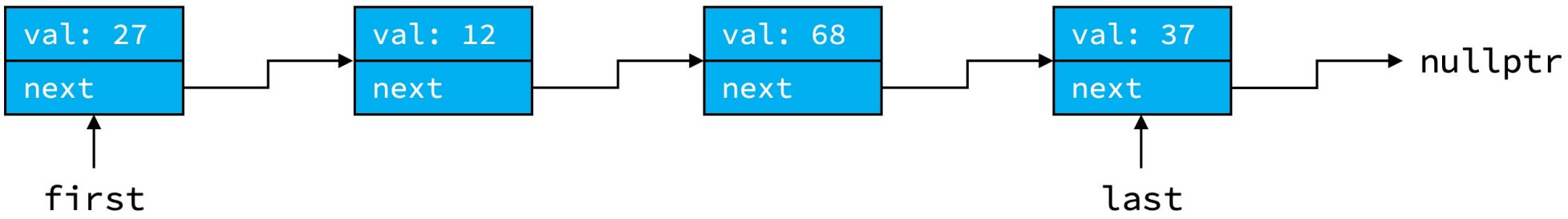
- Haben wir einen Zeiger auf eine Struktur, Klasse oder Union, brauchen wir **zwei** Operationen, um auf ein Mitglied zuzugreifen
  - Zuerst müssen wir den Pointer mit `*` dereferenzieren
  - Dann müssen wir mit `.` auf das entsprechende Mitglied zugreifen

```
Film* f = new Film;
(*f).titel = "Road_Trip";
```

- Mit dem Mitgliedszugriff-Operator für Zeiger `->` („member access operator“) kann man beide Schritte auf einmal erledigen

```
cout << f->titel << endl;
```

Programmieren!



- Schreibe ein C++-Programm das eine **verkettete Liste** für `int`-Werte mit den folgenden Operationen implementiert
  - `List* create()`
  - `void append(List* l, int val)`
  - `void remove(List* l, int val)`
  - `void print(List* l, ostream &os)`
  - `void destroy(List* l)`

Debugging

- Beim Hantieren mit Zeigern kann es immer mal wieder zu folgender Fehlermeldung kommen

```
segmentation fault (core dumped)
```

  - Wir wissen bereits, dass das bedeutet, dass wir auf Speicher zugegriffen haben, der unserem Programm gar nicht gehört
  - Um solche Fehler zu finden, haben wir heute `valgrind` gesehen
- Manchmal muss man aber ein Programm auch Schritt für Schritt durchgehen, um zu sehen, wo was schief läuft
- Der **GNU Debugger** `gdb` kann das und vieles mehr!
  - Programme Anweisung für Anweisung ausführen
  - Werte von allen möglichen Variablen ausgeben
  - Breakpoints setzen und zu Breakpoints springen
- **GTK** Um `gdb` zu nutzen, muss man das Programm wiederum mit der Option `-g` kompilieren!

Debugging

- Den **GNU Debugger** ruft man, wie folgt, auf

```
> gdb ./list
```

- Sobald `gdb` das Programm geladen hat, können am Prompt `(gdb)` u.A. folgende Kommandos eingegeben werden
  - `run` started das Programm
  - `bt` gibt nach einem Segmentation Fault die Stack Trace aus
  - `b` setzt einen Breakpoint
  - `d` löscht einen Breakpoint
  - `c` lässt das Programm weiterlaufen
  - `p` gibt den Wert von Variablen, Ausdrücke und Funktionsaufrufen aus
  - `n` führt die nächste Zeile vollständig aus
  - `s` steigt in eine Funktion ab
  - `f` führt die aktuelle Funktion bis zum Ende aus