DB IIB: DBMS-Implementierung

Übung 7: Platten, Mehr zu C++

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2021/22

http://www.informatik.uni-halle.de/~brass/dbi21/

Inhalt

Neue Corona-Regeln

Aus der EMail des Rektorats vom Montag:

- Die Lehre wird im Wintersemester weiterhin grundsätzlich in Präsenz durchgeführt.
- Allerdings werden ab 24. November 2021 nur noch Testnachweise einer öffentlichen Teststelle für Studierende und Lehrende akzeptiert.

  Die Möglichkeit des Nachweises durch Selbsttests entfällt damit.
- Die Maskenpflicht in den Lehrveranstaltungen wird ausgeweitet.

  Es gilt ab diesem Zeitpunkt eine FFP2-Maskenpflicht in allen Lehrveranstaltungen.
- Ebenso gilt, dass die Einhaltung der 3G-Regel in jeder Lehrveranstaltung mindestens stichprobenartig zu kontrollieren ist.

Inhalt

Aufgabe 5a (1)

Aufgabe:

- Finden Sie heraus, welche Platte oder SSD in Ihrem PC eingebaut ist.

Linux:

- `gnome-disks`
- `sudo smartctl -d ata -a -i /dev/sda`
- [https://www.cyberciti.biz/faq/find-hard-disk-hardware-specs-on-linux/]
- [https://unix.stackexchange.com/questions/4561/]

Aufgabe 5a (2)

Windows:

- Device Manager

  Settings→System→about, dann auf der Seite unten.

- `msinfo32`

Mac:

- Über diesen Mac → Systembericht → Festplatte

Aufgabe 5a (3)

Ergebnisse:

- 32 Abgaben, manche mit mehreren Platten.
- 7 Festplatten von Seagate

  ST2000LM015-2E8174, ST1000DM010-2EP1 (1 TB), ST500DM002-1BD142 (500 GB), ST2000DM008 (2 TB), ST1000LM035 (Seagate Mobile, 1 TB), ST2000DM008-2FR102, ST1000DM010-2EP102 (1 TB)
- 6 Festplatten von Western Digital

  WD Red Pro NAS-Festplatte 4TB

  Western Digital PC SN520 SDAPNUW-512G-1014

  wdc wd30ez30rx-00specb0 ($\rightarrow$? WD Caviar Green 3TB SATA,WD30EZRX)

  WD Red WD20EFRX (2TB 3.5" NAS Interne Festplatte)

  WD5000AAKX (Blue 500GB interne Festplatte)

  WDC WD10SPCX-24HWST1 (WD Blue 1 TB)

Aufgabe 5a (4)

Ergebnisse, Forts.:

- 3 Festplatten von Toshiba

  TOSHIBA DT01ACA100 (1 TB)
  TOSHIBA MD04ACA400 (4 TB)
  TOSHIBA DT01ACA200 (2 TB)

- 1 andere Festplatte

  LITEON LCH-512V2S

- Insgesamt also 17 Festplatten.
- Im Vergleich dazu 23 SSDs

  Siehe nächste Folien.

Aufgabe 5a (5)

Ergebnisse, Forts.:

- 9 SSDs von Samsung
  - Samsung 970 EVO Plus 1 TB NVMe SSD
  - Samsung 970 EVO 500GB NVMe
  - Samsung SSD 970 EVO 500GB
  - Samsung 870 EVO 1TB
  - Samsung SSD 860 EVO 250GB
  - Samsung SSD 850 EVO 250GB
  - Samsung SSD 850 EVO 500GB
  - Samsung SSD 850 Evo 120 GB
  - SAMSUNG MZFLW256HEHP-000MV
- 4 SSDs von Western Digital
  - Western Digital WD_BLACK SN750 NVMe SSD 500GB, M.2
  - WDS500G3X0C-00SJG0
  - WDS500G1X0E-00AFY0 (500 GB M.2 NVMe)
  - WDC WDS240G2G0A-00JH (240 GB)

Aufgabe 5a (6)

Ergebnisse, Forts.:

- 4 SSDs von SanDisk

  SanDisk SD8SN8U-128G-1006

  SanDisk SSD PLUS 240GB

  SanDisk SD9SN8WI128G1102

  SanDisk X400 128GB

- 6 andere SSDs

  M.2_PCIe-NVMe-SSD, 512GB

  Skhynix BC501 NVMe 128GB

  KBG30ZMS256G NVMe TOSHIBA 256GB

  APPLE SSD AP1024Q

  Solid State Drive

  SSD von Intenso

Aufgabe 5b (1)

Aufgabe:

- Für die folgende Aufgabe können Sie eine beliebige Magnetplatte wählen (keine SSD).
- Finden Sie zu der gewählten Platte möglichst viele Daten heraus, mindestens vier verschiedene Werte.

  Die Kapazität sollte ja einfach sein, auch die „Seek Time“ und die Rotationsgeschwindigkeit werden oft angegeben. Geben Sie auch die Quelle für die Daten an (z.B. Webadresse).

  Bei der „Seek Time“ können Sie vielleicht mehrere Datenwerte bekommen, z.B. der durchschnittliche Wert, sowie die Zeit für den Wechsel zu nächsten Spur, und vielleicht der Wert für die maximale Distanz von ganz innen nach ganz außen (oder umgekehrt).

  Die Geschwindigkeit für sequentielles Lesen wäre interessant.

  Auch Größen-Angaben wie die Anzahl der Oberflächen, die Anzahl der Spuren bzw. der Zylinder und die Anzahl der Sektoren pro Spur hätte ich gern.

Aufgabe 5b (2)

Links:

- [http://www.storagereview.com/]
- [http://www.tomshardware.com/]
- [https://www.seagate.com/internal-hard-drives/hdd/] (en)
  [https://www.seagate.com/de/de/internal-hard-drives/hdd/]
- [https://www.hgst.com/products/hard-drives] (en)
  [https://www.hgst.com/de/products/hard-drives] (de)
- [https://www.wdc.com/products/internal-storage.html] (en)
  [https://www.wdc.com/de-de/products/internal-storage.html]

Aufgabe 5b (3)

- Haben Sie sich `storagereview.com` `storagereview.com` angeschaut?
  A. Ja, nützlich/interessant.
  B. Ja, aber lohnt sich nicht.
  C. Nein.
- Gleiche Frage für `tomshardware.com`.
- Und noch einmal für irgendeine der drei Hersteller-Seiten.

Aufgabe 5b (4)

Links für Benchmarks:

- [https://www.heise.de/download/product/crystaldiskmark-46961]
- [https://crystalmark.info/en/software/crystaldiskmark/]
- [https://www.chip.de/downloads/ATTO-Disk-Benchmark_41802004.html]
- [https://www.atto.com/disk-benchmark/]
- [https://www.thomas-krenn.com/de/wiki/Fio]

Aufgabe 5b (5)

- Windows hat `winsat`, z.B. `winsat disk -drive d` oder `winsat disk -seq -v -read -drive c`

  Man muss eine Konsole als Administrator starten und es darin aufrufen.

  [http://technet.microsoft.com/en-us/library/cc770542.aspx]

- Beispiel-Ausgabe (aus einer Hausaufgaben-Abgabe):

```
Disk Random 16.0 Read                    1.20 MB/s   3.4
Disk Sequential 64.0 Read              105.44 MB/s   6.6
Disk Sequential 64.0 Write              95.53 MB/s   6.5
Avg. Read Time with Seq. Writes          3.244 ms   6.6
Latency: 95th Percentile                53.367 ms   1.9
Latency: Maximum                       113.955 ms   7.6
Avg. Read Time with Rand. Writes        18.171 ms   2.8
```

  Default für `-seqsize` ist 65536, Maximum ist 1 MB.

  Default für `-ransize` ist 16384, Maximum auch 1 MB.

Aufgabe 5b (6)

Seagate Barracuda ST5000LM000:

- Kapazität: 5 TB
- Datentransferrate: bis zu 140 MB/s
- Byte pro Sektor (logisch/physikalisch) 512/4096
- Schnittstelle: SATA mit 6 Gbit/s
- Cache: 128 MB
- Spindelgschwindigkeit: 5400 U/min
- Parken und Starten der Lese-/Schreibeköpfe: 600.000
- Nicht korrigierbare Lesefehler pro gelesenen Bits, max: 1 pro 10^{14}
- Preis (Amazon): 131.49 €

Aufgabe 5b (7)

Seagate ST1000DM010:

- [https://www.seagate.com/www-content/datasheets/pdfs/3-5-barracuda-3tbDS1900-10-1710US-en_US.pdf]

  [https://www.hdsentinel.com/storageinfo_details.php?lang=en&model=SEAGATE%20ST1000DM010]

- Kapazität: 1TB
- Average Seek Time: 8.5ms
- 1 Disk und 2 Heads
- Rotational Speed: 7200 RPM
  Rotation Time: 8.33 ms
  Average Rotational Latency: 4.17 ms
- Anschluss SATA 6 GB/s
- Stromverbrauch Leerlauf Durchschnitt: 4.6W

Aufgabe 5c (1)

Aufgabe:

- Schauen Sie sich die folgende Webseite an: [https://www.backblaze.com/blog/hard-drive-stats-for-2019/]
- Eine Übersicht der Daten für mehrere Jahre finden Sie hier: [https://www.backblaze.com/b2/hard-drive-test-data.html]
- Beantworten Sie folgende Frage: Wie viel Prozent der Platten fallen in einem Jahr ungefähr aus (größenordnungsmäßig)?

Aufgabe 5c (2)

- Ausgehend von der ersten Quelle mit Werten zu den Jahren 2017 bis 2019, fallen 1,25% bis 1,89% aller Platten aus.
- Schroeder&Gibson 2007 gehen in ihrem Paper sogar von 2-4% als üblicher Ausfallrate aus.

Inhalt

Fragen in mündlichen Prüfungen (1)

- Was sind die Haupt-Komponenten einer Festplatte?
- Was ist ein Sektor? Wie kann man einen Sektor auf der Festplatte adressieren?
- Was ist eine typische Sektorgröße? Warum verwenden viele DBMS und Betriebssysteme eine größere Blockgröße („Cluster-Größe“ in Windows)? Was ist eine typische Blockgröße? Diskutieren Sie Vor- und Nachteile einer größeren Blockgröße.
- Warum haben viele Platten mehr Sektoren auf den äußeren Spuren als auf den inneren Spuren?
- Ist es möglich, dass Blöcke bei einem Stromausfall nur partiell geschrieben werden?

Fragen in mündlichen Prüfungen (2)

- Warum ist „Read Ahead“ durch den Plattencontroller eine gute Idee?
- Wie lange braucht eine typische Platte ungefähr, um einen Block zu lesen oder zu schreiben? (Natürlich gibt es nicht nur eine korrekte Antwort, sondern einen ganzen Bereich. Sie sollten aber die Größenordnung kennen.)
- Was sind die drei Haupt-Komponenten der Zugriffszeit für einen Block?
- Wie viele wahlfreie Blockzugriffe kann eine typische Platte pro Sekunde schaffen?

Fragen in mündlichen Prüfungen (3)

- Wie viele Byte kann man von einer typischen Platte pro Sekunde sequentiell lesen?
- Was sind typische Rotationsgeschwindigkeiten von Platten? Welchen Teil der Zugriffszeit für einen Block beeinflusst die Rotationsgeschwindigkeit hauptsächlich?

Inhalt

Arrays (1)

- Mathematisch gesehen ist der Wert eines Arrays eine Abbildung vom Indexbereich auf den Wertebereich des Element-Datentyps, im Beispiel:

$$\{0, 1, 2, 3, 4\} \rightarrow \texttt{int}: \{-2^{31}, \ldots, 2^{31} - 1\}$$

- Man kann sie als Tabelle darstellen (Beispiel):

Index	Inhalt
0	27
1	42
2	18
3	73
4	56

Arrays (2)

- Für ein Array der Größe n reserviert der Compiler den n-fachen Speicherplatz wie für eine einzelne Variable des entsprechenden Typs.
- Wenn z.B. ein `int` 4 Byte belegt, reserviert der Compiler für das Array `a`: $5 * 4 = 20$ Byte.

  Man kann das überprüfen, indem man `sizeof(int)` und `sizeof(a)` ausgibt. Diese Beziehung gilt immer.
- Wenn das Array z.B. ab Adresse 1000 beginnt, steht an dieser Stelle der Wert von `a[0]`.

  Er belegt die vier Bytes mit den Adressen 1000 bis 1003.

Arrays (3)

- Ab Adresse 1004 steht dann `a[1]`.
- Die fünf `int`-Werte stehen also direkt hintereinander im Speicher:

1000:	a[0] = 27
1004:	a[1] = 42
1008:	a[2] = 18
1012:	a[3] = 73
1016:	a[4] = 56

27, 42, 18, 73, 56 sind hier irgendwelche (sinnlosen) Beispiel-Inhalte des Arrays (Variablenwerte).

Arrays (4)

- Allgemein kann man mit

  $$T\ v[n];$$

  ein Array v der Größe n für Werte des Typs T deklarieren. Die Größe n muß zur Compilezeit bekannt sein (also nicht aus Eingaben etc. berechnet).

- Die Variable v hat dann den Typ $Array(n, T)$.

  Dies ist meine private (theoretische) Schreibweise für Datentypen. So kann man es nicht in C/C++ aufschreiben.

- Hat T die Größe g, so hat der Typ $Array(n, T)$ die Größe $n * g$.

Arrays (5)

- Hat der Compiler für v einen entsprechend großen Speicherbereich ab Adresse s belegt, so steht das Array-Element $v[i]$ an Adresse $s + i * g$.

  Deswegen ist es einfacher, wenn der Indexbereich mit 0 beginnt.

- Der Compiler erzeugt zum Zugriff auf Arrayelemente Maschinenbefehle, die diese Adressberechnung zur Laufzeit vornehmen.

  Viele CPUs haben Adressierungsmodi, die die diese Berechnung etwas vereinfachen/beschleunigen. Die Multiplikation mit der Arraygröße muß aber wohl explizit durchgeführt werden. Bei Zweierpotenzen kann der Compiler natürlich einen Shift-Befehl verwenden.

Arrays (6)

- Der Arrayzugriffsoperator `_[_]` hat die Signatur

$$Array(n, T) \times \texttt{int} \rightarrow Lvalue(T)$$

  Tatsächlich arbeitet er mit Pointern, s.u.

- Statt `int` können alle ganzzahligen Typen verwendet werden, z.B. auch `long` oder `unsigned int`.

- Der Programmierer ist dafür verantwortlich, daß er nur in den definierten Grenzen auf ein Array zugreift. Bei $v[i]$ muß i einen Wert zwischen 0 und $n-1$ haben (wenn n die Größe des Arrays ist): s.u.

Arrays (7)

- Im Innern der eckigen Klammern `[...]` kann ein beliebig komplizierter Wertausdruck stehen, der einen ganzzahligen Typ liefert, z.B.

```
a[(i*2+1) % 5]
```

- Da das Ergebnis des Array-Zugriffs ein Lvalue (eine Variable) ist, kann es auch auf der linken Seite einer Zuweisung stehen:

```
a[i] = 27;
```

- Wie üblich, wird *Lvalue*(*T*) bei Bedarf in *T* umgewandelt (durch Zugriff auf den Variablen-Inhalt).

Arrays (8)

- Man kann daher z.B. den Inhalt der Array-Elemente so ausgeben:

```
for(int i = 0; i < 5; i++)
        cout << "a[" << i << "] = "
                        << a[i] << "\n";
```

- Natürlich müssen alle abgefragten Arrayelemente vorher initialisiert sein (wie bei allen Variablen).

  Es ist möglich, daß nur ein Teil der Indexpositionen initialisiert sind, und andere nicht. Auf die nicht initialisierten Indexpositionen darf man dann nicht lesend zugreifen.

- Häufig werden `for`-Schleifen zum Zugriff auf die Array-Elemente benutzt.

Arraygrenzen-Verletzung (1)

- Sei das Array `a` wieder deklariert als: `int a[5];`
- Beim Array-Zugriff `a[i]` muß `i` einen Wert zwischen `0` und `4` (inklusive) haben.
- Ist das nicht der Fall, kann folgendes passieren:
  - Das Programm wird zur Laufzeit mit einem Fehler abgebrochen ("array index out of bounds", "segmentation fault", etc.).
  - Es wird auf die gemäß der Formel $s + i * g$ berechnete Adresse zugegriffen, dort steht aber nicht ein Array-Element, sondern andere Daten.

Arraygrenzen-Verletzung (2)

```
...
int main()
{
        int n = -111;
        int a[5];
        int m = -222;

        for(int i = 0; i < 5; i++)
                a[i] = 10 * i;

        for(int j = -1; j < 6; j++)
                cout << j << ": " << a[j] << "\n";

        return 0;
}
```

Arraygrenzen-Verletzung (3)

- Auf meinem Rechner mit Visual C++ 6.0 ergibt sich folgende Ausgabe (nur ohne Tabellenrahmen):

j	a[j]	
-1	-222	⟵ m
0	0	
1	10	
2	20	
3	30	
4	40	
5	-111	⟵ n

- Der Compiler hat also die Variable `m` vor dem Array `a` im Speicher abgelegt, und `n` dahinter.

Arraygrenzen-Verletzung (4)

- Man kann die Adressen der Variablen auch mit dem Adressoperator `&` abfragen (s.u.), dies bestätigt die Vermutung:

Variable	Adresse
`n`	7077364
`a`	7077344
`m`	7077340

- C++ schreibt nicht vor, wie der Compiler die Variablen im Speicher anzuordnen hat. Man kann sich also keineswegs darauf verlassen, daß man mit `a[-1]` auf `m` zugreifen kann.

Arraygrenzen-Verletzung (5)

- Probiert man auf `a[1000]` zuzugreifen, so erhält man (unter Windows):

  Arrtest has caused an error in ARRTEST.EXE
  Arrtest will now close.

- Klickt man auf Debuggen, so wird der Laufzeitfehler genauer angezeigt:

  Unhandled Exception in arrtest.exe:
  0xC0000005: Access Violation

- Unter UNIX bekommt man: "Segmentation Fault".

- Die für `a[1000]` berechnete Adresse liegt außerhalb des Bereichs, der diesem Prozess zugeordnet ist.

Arraygrenzen-Verletzung (6)

- Besonders übel sind Schreibzugriffe mit ungültigen Wert für den Array-Index.
- Im Beispiel würde

```
a[-1] = 1;
```

  den Wert von `m` ändern, ohne daß im Programm eine Zuweisung an `m` steht.
- Solche Fehler sind schwierig zu finden.

  Man braucht oft viel Zeit. Manchmal scheint es fast hoffnungslos. Gute Debugger haben eine Funktion, mit der man eine Speicherstelle auf Änderungen überwachen kann.

Arraygrenzen-Verletzung (7)

- Bei Zugriffen über einen ungültigen Arrayindex werden natürlich auch die Datentypen nicht beachtet. Der Compiler nimmt ja an, daß ein Integer geschrieben wird. Hat `m` aber z.B. den Typ `float`, so hat es nach `a[-1] = 1` den Wert `1.401298e-45`.
- Es ist auch möglich, daß die Rücksprungadresse bei Prozeduren oder die Werte von temporär gesicherten Registern überschrieben werden.
- Dann kann man sich auf gar nichts mehr verlassen: Alles ist möglich.

Arraygrenzen-Verletzung (8)

- Ob in einem Programm jemals ein Zugriff auf ein Array-Element außerhalb der Grenzen vorkommen kann, ist wieder eine unentscheidbare Frage.
- Es kann also keinen Compiler geben, der bereits bei der Übersetzung des Programms diesen Fehler findet.
- Er könnte allerdings Maschinencode erzeugen, der zur Laufzeit vor jedem Arrayzugriff den Index prüft und ggf. das Programm mit einer Fehlermeldung abbricht (“array index out of bounds”).

Arraygrenzen-Verletzung (9)

- So ein Test vor jedem Array-Zugriff bringt aber einen nicht unwesentlichen Overhead mit sich.

  Verlangsamung des Programms, ggf. zusätzlich nötiger Speicher.

- Daher bauen die wenigsten Compiler solche Tests in den Maschinencode ein.

  Eventuell gibt es eine Option dafür. Es hängt auch von der Programmiersprache ab, ob die Compilerschreiber solch einen Test als wichtig ansehen. Z.B. sind solche Tests bei Java üblich, wohl auch bei Pascal.

- In C/C++ ist das Problem dadurch verschärft, daß es auch Zeiger (Pointer) gibt, und der Array-Zugriff eigentlich als Pointer-Zugriff behandelt wird (s.u.).

Arraygrenzen-Verletzung (10)

- Das folgende Programmstück ist unsicher:

```
char input[80];
int i = 0; char c;
while(cin.get(c) && c != '\n')
        input[i++] = c;
input[i] = 0; // Ende-Markierung, s.u.
```

`cin.get(c)` liest das nächste Eingabezeichen in die Variable `c`. Es liefert den Stream, bei der Umwandlung in einen boolschen Wert wird sein `fail()`-Status abgefragt.

- Enthält die Eingabezeile weniger als 79 Zeichen, funktioniert es. Ist sie aber länger, werden Speicherzellen hinter dem Array überschrieben.

Arraygrenzen-Verletzung (11)

- Leider enthalten viele alte Programme solchen Programmcode, bei dem es zu einem “Buffer Overflow” kommen kann.
- Wenn die Array-Größe hinreichend groß gewählt wird, geht bei nicht böswilligen Benutzern ja auch alles gut.
- Hacker nutzen die Sorglosigkeit des Programmierers aber aus, indem sie mit zu langen Eingaben beliebig Speicherbereiche überschreiben.

Arraygrenzen-Verletzung (12)

- Wenn die Hacker den String geschickt basteln, ist es möglich, beliebige Befehle auszuführen, die so gar nicht im Programm stehen.

  Z.B. könnte die Rücksprungadresse überschreiben werden, so daß der Rücksprung in das Array erfolgt, in das der Hacker beliebige Maschinenbefehle eingetragen hat. Falls der Speicherbereich nicht ausgeführt werden kann (manche CPUs haben ein "No-Execute" Bit für Speicherseiten), kann man auch einen "Rücksprung" basteln, der wie ein Aufruf einer Bibliotheksfunktion wirkt — mit beliebigen Eingabewerten (z.B. kann man mit "`system`" ein beliebiges Programm aufrufen).

- Man sollte unbedingt so programmieren, daß Arraygrenzen unter keinen Umständen verletzt werden können.

Konstanten (Ausblick)

- Oft steht die Array-Länge an mehreren Stellen im Programm (z.B. Deklaration, Eingabe-Begrenzung).
- Es ist dann besser, eine Konstante zu deklarieren:

```
const int laenge = 80;
```

- Das Array kann dann so deklariert werden:

```
char input[laenge];
```

Mit einer gewöhnlichen Variable würde das (zumindest nach dem Standard) nicht funktionieren, da die Array-Länge zur Compilezeit bekannt sein muss. Der GNU-Compiler hat eine Erweiterung gegenüber dem Standard, so dass es dort auch mit Variablen geht, aber das könnte dann mit anderen Compilern zu Fehlern führen (nicht portabel, schlecht).

Vektor (Ausblick)

- Die Standard-Bibliothek von C++ bietet eine parametrisierte Klasse `vector<T>` an, die einem Array mit Elementtyp T recht ähnlich ist, aber man kann
  - die Größe zur Laufzeit wählen, und sogar noch nachträglich verändern,

    Um die Klasse zu verwenden, muss man oben `#include <vector>` schreiben, und den Namespace `std` nutzen (wie für `cout`). Dann kann man z.B. `vector<int> a(5);` deklarieren. Die Elemente sind automatisch auf 0 initialisiert.
  - auch mittendrin Elemente einfügen oder löschen,
  - beim Zugriff den Index-Wert überprüfen lassen.

    Bei `a[i]` wird nicht überprüft, bei `a.at(i)` schon.

Zeiger (1)

- `int *p;` deklariert eine Variable `p`, die die Hauptspeicheradresse einer `int`-Variable aufnehmen kann.
- Der Wert von `p` ist ein Zeiger/Pointer (auf ein Integer). Wir schreiben $Pointer(\texttt{int})$ für den Typ von p.
- Mit dem Operator `&` kann man die Adresse einer Variablen bestimmen. Er hat die Signatur

$$Lvalue(T) \rightarrow Pointer(T)$$

- Umgekehrt erlaubt $*$ den Zugriff auf die Variable, auf die ein Pointer zeigt:

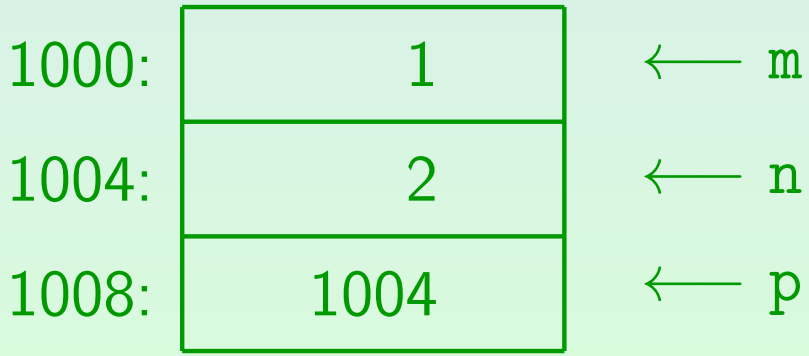
$$Pointer(T) \rightarrow Lvalue(T)$$

Zeiger (2)

- Beispiel:

```
int m = 1;
int n = 2;
int *p = &n;
```

- Falls die Variablen die Adressen 1000, 1004, 1008 haben, ist die Situation im Speicher:



Zeiger (3)

- Die Zuweisung

```
*p = 5;
```

  ändert indirekt den Wert der Variablen `n`:

- Da `p` die Adresse von `n` enthält, bewirkt die obige Zuweisung das gleiche wie

```
n = 5;
```

- Es macht Programme natürlich schwerer zu verstehen, wenn man gleichzeitig über verschiedene Namen (aliases) (`n` und `*p`) auf die gleiche Speicherstelle zugreifen kann: besser vermeiden.

Zeiger (4)

- Die Operation * (Übergang vom Pointer auf die Variable und ggf. weiter auf den Wert) nennt man auch Dereferenzierung.

  Der Name ist in C++ eventuell etwas unglücklich, da es außer Zeigern auch Referenzen gibt, die automatisch dereferenziert werden, s.u.

- Wie bei Arrayzugriffen außerhalb des Indexbereichs kann die Zuweisung über einen Pointer, der nicht auf eine Variable des entsprechenden Typs zeigt, den Inhalt beliebiger Speicherzellen überschreiben.

  Besonders übel sind Zuweisungen über nicht-initialisierte Pointer.

Null-Zeiger

- Zeiger können den Wert 0 enthalten (“Nil”).

  Es ist nicht garantiert, daß das intern das gleiche Bitmuster ist.

- Dieser Wert wird verwendet, um auszudrücken, daß der Zeiger nirgendwohin zeigt.

- C/C++ garantieren, daß niemals eine Variable an dieser Adresse abgelegt wird.

  Damit kann man den Null-Zeiger sicher von allen legalen Speicheradressen unterscheiden.

- Der Versuch, einen Null-Zeiger zu dereferenzieren, führt zu einem Laufzeit-Fehler (“access violation” bzw. “segmentation fault”).

Deklarations-Syntax (1)

- In C/C++ schreibt man auf die rechte Seite der Deklaration einen Ausdruck, der den Typ auf der linken Seite liefern würde:
  - `*p` ist ein `int`:
    Dann muß `p` ein Zeiger/Pointer auf `int` sein.
  - `a[5]` ist ein `int`:
    Dann muß `a` ein Array von `int`-Elementen sein.

    Eigentlich wäre es logisch, wenn das Array dann die Größe 6 hat, so daß `a[5]` wirklich noch legal ist. Aber die C-Entwickler glaubten wohl, daß das noch verwirrender wäre, als die Zahl direkt als Array-Größe zu verwenden.

Deklarations-Syntax (2)

- Ob und wo man in der Deklaration einer Zeiger-Variable ein Leerzeichen einfügt, ist egal:
  - `int *p;` (üblich)
  - `int* p;` (möglich, um den Typ zu betonen)
  - `int*p;` (legal, aber nicht hübsch)
- Deklariert man aber zwei Zeiger-Variablen gemeinsam, muss man den * bei jeder Variable schreiben:

```
int *p, *q;
```

Bei "`int* p, q;`" wäre `q` nur ein `int`, und kein Zeiger.

Zeiger und Const: Ausblick

- Man beachte, daß String-Konstanten, z.B. `"abc"` nicht geändert werden dürfen.

  Der Compiler kann sie in einem schreibgeschützten Bereich ablegen.

- Einen Zeiger auf eine String-Konstante muß man deklarieren mit

  ```
  const char *p;
  ```

- In diesem Fall ist nicht der Zeiger konstant, sondern `*p`, also der Bereich, auf den der Zeiger zeigt.

  Eine Zuweisung an die Speicherstelle in `p`, z.B. `*p = 'x';` wäre dann also verboten, nicht aber die Änderung des Inhalts der Variablen `p` (z.B. Weiterschalten auf nächstes Zeichen: `p++`).

Untypisierte Zeiger

- Manchmal sind Zeiger nur Hauptspeicher-Adressen, und zeigen nicht auf Variablen bestimmten Typs.

  Z.B. wenn man eine eigene Speicherverwaltung programmiert.

- Man verwendet dafür den Typ `void*`, weil es keine Variablen vom Typ `void` gibt.

  `void` ist der "leere Typ", der als Rückgabetyp von Prozeduren eingeführt wurde, die keinen Wert zurückliefern.

- Wenn `p` den Typ `void*` hat, und `q` z.B. `int*`, ist die Zuweisung `p = q;` möglich, aber die umgekehrte Richtung geht nur mit expliziter Typumwandlung:

  `q = (int*) p;` *oder* `q = static_cast<int*>(p);`

Zeiger und Arrays (1)

- Eine wichtige Anwendung von Zeigern ist es, Array-Zugriffe etwas zu beschleunigen.

  Man spart oft die Multiplikation zur Adress-Berechnung. Der Programmtext wird in der regel auch etwas kürzer.

- In C/C++ gibt es einen ganz engen Zusammenhang zwischen Arrays und Zeigern:
  - Der Typ $Array(n, T)$ wird praktisch überall automatisch in den Typ $Pointer(T)$ umgewandelt.
  - Schreibt man einfach den Array-Namen, so ist dies die Basisadresse (Startadresse) des Arrays.

Zeiger und Arrays (2)

- Der Additions-Operator kann auch zur Adressberechnung für Array-Elemente genutzt werden. Er hat auch die Signatur

$$\mathit{Pointer}(T) \times \texttt{int} \rightarrow \mathit{Pointer}(T)$$

- Wenn `p` die Basis-Adresse des Arrays ist, so ist `p+1` die Adresse des Elements an Indexposition 1 (zweites Element), `p+2` die Adresse von `p[2]`, u.s.w.
- Bei der Addition wird also die Größe des Typs T berücksichtigt: Falls T z.B. `int` (4 Byte) ist, addiert `p+1` tatsächlich 4 auf die Adresse in `p`.

Zeiger und Arrays (3)

- In C/C++ ist `a[i]` nur eine Abkürzung für

  `*(a+i)`.

- Der Additions-Operator ist auch in der Pointer-Variante kommutativ, er hat also auch die Signatur

  $\texttt{int} \times \mathit{Pointer}(T) \rightarrow \mathit{Pointer}(T)$.

- Daher kann man in C/C++ statt `a[i]` auch `i[a]` schreiben, das ist definitionsgemäß äquivalent.

- Nicht alles was geht, ist auch gut! Man sollte sich an die üblichen, allgemein bekannten Muster halten.

Zeiger und Arrays (4)

- Auch die Vergleichsoperatoren `==`, `!=`, `<`, `<=`, `>`, `>=` sind für Zeiger definiert:

$$Pointer(T) \times Pointer(T) \rightarrow \texttt{bool}$$

  Es reicht, wenn die beiden Pointer auf kompatibele Typen zeigen.

- Das Ergebnis ist aber nur dann wohldefiniert, wenn beide Zeiger in das gleiche Array (oder das gleiche Objekt/die gleiche Struktur) zeigen.

  Manche CPUs verwenden Zeiger innerhalb verschiedener Speichersegmente.

- Zeigt z.B. `p` auf `a[1]` und `q` auf `a[3]`, so gilt `p < q`.

Zeiger und Arrays (5)

- Es ist legal, wenn bei der Adressberechnung ein Zeiger herauskommt, der direkt hinter ein Array zeigt.

  Also auf das nicht existierende Element `a[`n`]` bei einem Array der Größe n.

- Man darf diesen Zeiger nicht dereferenzieren, aber `+` gibt keinen Überlauf, und Vergleiche mit Zeigern auf andere Array-Elemente sind definiert.

- Entsprechend zum Operator `+` können auch Adressen mit dem Operator `-` berechnet werden:

  - Wenn `p` z.B. auf `a[4]` zeigt, so ist `p-3` die Adresse von `a[1]`.

Zeiger und Arrays (6)

- Der Operator `-` kann aber auch den Abstand zwischen zwei Array-Elementen berechnen:
  - Hierfür hat er die Signatur

    $$Pointer(T) \times Pointer(T) \rightarrow \texttt{int}$$

    Genauer liefert er `ptrdiff_t`, das in `<stddef.h>` bzw. `<cstddef>` als `int` oder als `long` definiert ist.
  - Wenn p z.B. auf `a[4]` zeigt, und q auf `a[1]` zeigt, dann liefert $p - q$ den Wert 3.

    Die tatsächliche Differenz der Adressen wird dabei durch die Größe der Array-Elemente geteilt, bei `int`-Werten also z.B. durch 4.

Zeiger und Arrays (7)

- Auch die Inkrement- und Dekrement-Operatoren (`++`, `--`) funktionieren auf Zeigern:

$$Lvalue(Pointer(T)) \rightarrow Pointer(T).$$

  Bei `++p` und `--p` kommt tatsächlich auch ein *Lvalue* heraus.

- Wenn `p` z.B. auf `a[2]` zeigt, so zeigt es nach `p++` auf `a[3]`.

  Wieder wird hier nicht unbedingt 1 auf die Adresse in `p` addiert, sondern die Größe des Typs T.

- Damit kann man gut Schleifen programmieren, die über ein Array laufen.

Zeiger und Arrays (8)

- Beispiel (Länge eines C-Strings bestimmen):

```
char s[100];
...// s fuellen
for(char *p = s; *p; p++)
        ; // Leerer Rumpf
int laenge = p - s;
```

- Aufgabe: Was halten Sie von dieser Variante?

```
p = s;
while(*p++);
int laenge = p - s;
```

- Und was ist mit `while(*++p)`?

Zeiger und Arrays (9)

- Wenn man die Zeigerdifferenz nicht mag, kann man auch einen Zähler mitlaufen lassen:

```
int laenge = 0;
for(char *p = s; *p; p++)
        laenge++;
```

- Aufgabe: Würde dies auch funktionieren?

```
char *p = s;
int laenge = 0;
while(*p++)
        laenge++;
```

Zeiger: Vergleich mit Pascal

- C/C++ bietet den vollen Zugriff auf alle Möglichkeiten der CPU (für Benutzerprozesse).
- Das ist gut für die Effizienz.

  In anderen Sprachen braucht man einen sehr guten Codeoptimierer.
- Andererseits bietet das viel mehr Möglichkeiten für schwer zu findene Fehler.
- In Pascal gibt es auch Pointer, aber sie werden nur zur dynamischen Speicherverwaltung benutzt (s.u.).

  Man kann nicht die Adresse einer Variablen bestimmen (kein &) und kann nicht mit Adressen rechnen (kein +, - für Pointer). Es gibt auch keine Beziehung zwischen Arrays und Pointern.