Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 8: Arrays und Zeiger (Pointer)

- Deklaration von Arrays, Speicherbelegung
- Zugriff auf Array-Elemente, Adressberechnung
- C-Strings, Klasse string
- Deklaration von Zeigern, Dereferenzierung, Adressbestimmung, Zeigerarithmetik

Inhalt

1. Einführung, Arraygrenzen-Verletzung
2. C-Strings, Klasse string
3. Initialisierung von Arrays
4. Mehrdimensionale Arrays
5. Zeiger (Pointer): Einführung
6. Zeiger und Arrays, Zeiger-Arithmetik

Arrays (1)

- Bisher können Variablen nur einen einzigen Wert (z.B. eine Zahl, ein Zeichen) enthalten.
- Für viele Berechnungen muß man sich aber eine ganze Anzahl von Werten merken. Beispiele:
  - Zeichenketten bestehen aus mehreren Zeichen.
  - Wenn man mit dem "Sieb des Eratosthenes" Primzahlen bis zu einer Obergrenze n bestimmen will, muß man sich für alle Zahlen bis n merken, ob man schon einen Teiler gefunden hat.
  - Spielbretter (z.B. Schach, Sudoko).

Arrays (2)

- Arrays bestehen aus einer Menge von gleichartigen Variablen, die über eine Zahl (den Index) unterschieden werden.

  Man kann aber auch das ganze Array als eine Variable auffassen.
- Deutsch sagt man auch "Feld" oder "Vektor".

  Mehrdimensionale Arrays wären entsprechend Matrizen.
- Z.B. kann man mit

  ```
  int a[5];
  ```

  ein Array bestehend aus Speicherplätzen für fünf `int`-Werte deklarieren.

Arrays (3)

- Die einzelnen Speicherplätze kann man mit `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]` ansprechen.
- In C/C++ beginnt der Indexbereich immer mit 0, daher endet er für ein Array der Größe n mit $n-1$.
- In Pascal gibt man dagegen Untergrenze und Obergrenze des Indexbereiches explizit an:

```
a: array[0..4] of integer;   { Kein C! }
```

Man kann dort die Grenzen beliebig wählen, z.B.

```
a: array[1..5] of integer;   { Kein C! }
```


Arrays (4)

- Natürlich konnte man sich auch bisher schon fünf `int`-Variablen `a0`, `a1`, `a2`, `a3`, `a4` deklarieren.
- Will man sie aber z.B. alle auf 0 setzen, muss man für jede Variable einzeln eine Anweisung schreiben:

```
a0 = 0; a1 = 0; a2 = 0; a3 = 0; a4 = 0;
```

- Das Neue bei einem Array ist nun, dass man die Zahl, die gewissermaßen Bestandteil des Variablennamens ist, berechnen kann:

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


Arrays (5)

- 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 (6)

- 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 (7)

- 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 (8)

- Allgemein kann man mit

$$T\ v\texttt{[}n\texttt{];}$$

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 (9)

- 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 (10)

- 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 (11)

- 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 (12)

- 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). Konstanten werden in Kapitel 10 ausführlich behandelt.

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.

Inhalt

1. Einführung, Arraygrenzen-Verletzung
2. C-Strings, Klasse string
3. Initialisierung von Arrays
4. Mehrdimensionale Arrays
5. Zeiger (Pointer): Einführung
6. Zeiger und Arrays, Zeiger-Arithmetik

C Strings (1)

- Strings (Zeichenketten) werden in C als Arrays von Zeichen dargestellt, die mit einem Null-Zeichen abgeschlossen sind.
- `"abc"` ist im Speicher also als vier Bytes dargestellt:

`'a'`	`'b'`	`'c'`	`'\0'`

- Oder, wenn man die Bitmuster als Zahlen (ASCII-Codes) sieht:

97	98	99	0

C Strings (2)

- Es gibt in C also keinen speziellen (zusätzlichen) Typ für Zeichenketten, sondern `"abc"` hat einfach den Typ $\mathit{Array}(4, \texttt{char})$.

  Genauer gesagt ist es ein konstantes Array: Man kann den Inhalt nicht mit einer Zuweisung überschreiben. Siehe unten.

- Man kann den String z.B. ausgeben mit:

```
for(int i = 0; i < 3; i++)
        cout << "abc"[i];
```

  (natürlich funktioniert `<<` auch direkt für Strings, dies soll nur zeigen, daß `"abc"` formal ein Array ist).

C Strings (3)

- Es ist eine häufige Quelle von Fehlern, daß man das zusätzlich nötige Array-Element für die Ende-Markierung `'\0'` vergisst.
- Wenn man z.B. Namen bis zur Länge 20 als C-String in ein Zeichenarray speichern will, muß das Array die Größe 21 haben.
- Übliche Konvention ist, daß die Bytes hinter dem Null-Zeichen nicht abgefragt werden.

  Falls in das Array der Größe 21 also ein Name mit 4 Zeichen gespeichert wird, brauchen nur die ersten 5 Array-Elemente initialisiert zu sein. Im Rest des Arrays kann beliebiger Müll stehen.

C Strings (4)

- Beispiel (Bestimmung der Länge eines Strings):

```
int laenge = 0;
char string[80];
... // string einlesen etc.
while(string[laenge])
        laenge++;
```

- Weil 0 als logisch falsch gilt, hört die Schleife auf, sobald die Ende-Markierung ereicht ist.
- Da die Index-Positionen mit 0 beginnen, ist die Position der Ende-Markierung die Länge des Strings.

C Strings (5)

- Die Eingabe von C-Strings mit >> ist unsicher: In C/C++ wird an eine Funktion nur die Anfangsadresse des Arrays übergeben, der Operator >> weiß also nicht, wie lang das Array ist.
- Bei der Funktion `getline` kann man die Größe des Arrays mit übergeben, diese Variante ist sicher:

```
char input[80];
cin.getline(input, 80);
```

Im Unterschied zu >> werden Leerzeichen nicht übersprungen. Man kann als drittes Argument ein Zeichen übergeben, bei dem `getline` aufhören soll. Wenn man nichts angibt, ist das `'\n'`. Dieses Zeichen wird aus der Eingabe weggelesen, aber nicht in den String geschrieben.

Arrays und Zuweisung (1)

- Man kann Arrays nicht mit einem Befehl zuweisen:

```
int a[5];
int b[5];
... // Initialisierung von a
b = a; // Fehler!
```

- In C sollte es in der Regel möglich sein, elementare Operationen der Sprache (wie Zuweisungen) durch einen Maschinenbefehl auszuführen.
- Wenn etwas viele Maschinenbefehle benötigt, sollte der Programmierer es auch merken (indem er eine Schleife programmieren muß etc.).

Arrays und Zuweisung (2)

- Selbstverständlich kann ein Array kopiert werden, indem die Elemente einzeln zugewiesen werden:

```
for(int i = 0; i < 5; i++)
        b[i] = a[i];
```

- Für C-Strings gibt es eine Funktion `strcpy` (deklariert in `<string.h>` bzw. `<cstring>`) zum Kopieren der `char`-Arrays: `strcpy(⟨Ziel⟩, ⟨Quelle⟩);`

Aber man kann eben nicht direkt eine Zuweisung aufschreiben. Wenn man einen Funktionsaufruf macht, ist dem Programmierer bewußt, daß dies keine elementare Operation mehr ist und eventuell längere Zeit benötigt. Selbstverständlich kann man sich solche Funktionen auch für andere Arten von Arrays schreiben.

Arrays und Zuweisung (3)

- Entsprechend ist auch ein Test auf Gleichheit nicht auf ganzen Arrays möglich:

```
int a[5];
int b[5];
... // Initialisierung von a und b
if(a == b) ... // Fehler!
```

- Man kann dies z.B. so erreichen:

```
bool equal = true;
for(int i = 0; i < 5 && equal; i++)
        if(a[i] != b[i])
                equal = false;
if(equal) ...
```


Arrays und Zuweisung (4)

- Den Gleichheitstest auf Arrays kann man auch so programmieren:

```
int i = 0;
while(i < 5 && a[i] == b[i])
        i++;
if(i == 5) ...
```

- Für C-Strings gibt es eine Funktion `strcmp` zum Vergleich von Strings ("string compare").

  Sie liefert 0 (logisch falsch!), wenn die Zeichenketten gleich sind. Ein negativer Wert bedeutet, daß die erste Zeichenkette lexikographisch vor der zweiten kommt, bei positivem Wert ist es umgekehrt. (Allg.: Differenz der Zeichencodes an erster Position mit Unterschied.)

Arrays und Zuweisung (5)

- In C++ wurde die Sprachphilosophie in dieser Hinsicht grundlegend geändert:
  - ◇ Für Klassen kann man Operatoren wie `=` und `==` mit einer beliebigen Prozedur hinterlegen.
  - ◇ Dann funktionieren Zuweisung/Vergleich, auch wenn diese Klassen komplexe Datenstrukturen inklusive Arrays enthalten.
  - ◇ Manche Prozeduren werden sogar ganz ohne expliziten Aufruf ausgeführt (Konstruktoren, Destruktoren, Typumwandlungen).

Exkurs: Strings in C++ (1)

- Natürlich funktionieren C-Strings auch in C++.
- Die Standardbibliothek von C++ enthält aber auch eine Klasse `string`, die den Umgang mit Strings vereinfacht (gegen einen gewissen Effizienzverlust).
- Um sie nutzen zu können, benötigt man

```
#include <string>
```

- Insbesondere kümmert sich diese Klasse automatisch um die Speicherverwaltung, fordert also intern ein hinreichend großes Array von Zeichen an.
- Damit funktioniert `>>` wieder sicher.

Exkurs: Strings in C++ (2)

- Eine Variable “`s`” vom Typ `string` (ein Objekt dieser Klasse) kann man anlegen mit

```
string s;
```

- Wie erwartet kann man einen String mit >> einlesen und mit << ausgeben.
- Die aktuelle Länge des Strings “`s`” kann man abfragen mit

  `s.length()` oder äquivalent `s.size()`

  Dies sind Beispiele für Methoden-Aufrufe, die in einem späteren Kapitel genauer diskutiert werden.

Exkurs: Strings in C++ (3)

- Auf einzelne Zeichen kann man zugreifen wie bei einem Array mit

```
s[i]
```

- Wie bei einem Array werden die Index-Grenzen hier nicht geprüft. Will man das (sehr zu empfehlen, s.o.), muß man folgendes verwenden:

```
s.at(i)
```

  Greift man außerhalb der Grenzen auf den String zu, wird eine Exception (Ausnahme, Laufzeitfehler) ausgelöst, die normalerweise das Programm beendet. Man kann sie aber abfangen, wie in einem späteren Kapitel erläutert wird.

Exkurs: Strings in C++ (4)

- Die Zuweisung ist für Objekte der Klasse `string` definiert, man kann sogar einen C-String einem solchen Objekt zuweisen:

```
s = "abc";
```

- Zur Initialisierung verwendet man besser die Syntax

```
string s("abc");
```

- Entsprechend sind Vergleiche für `string`-Objekte möglich, auch Vergleiche mit C-Strings, z.B.

```
if(s == "abc") ...
```

  Auf einer Seite muß aber ein `string`-Objekt stehen (egal ob links oder rechts oder auf beiden Seiten).

Exkurs: Strings in C++ (5)

- Mit der `string`-Klasse kann man Strings auch konkatenieren (aneinanderhängen), z.B.

```
string s("abc");
s += "def";
```

  `s` hat nun den Wert `"abcdef"`.

- Sind `name`, `vorname`, `nachname` Objekte der Klasse `string`, geht auch

```
name = vorname + " " + nachname;
```

- Das ist allerdings nicht besonders effizient.

Exkurs: Strings in C++ (6)

- Die Klasse `string` bietet u.a. auch folgende Methoden (siehe Handbuch):
  - ◇ `s.substr(2, 4)`: Teilstring ab Index 2, Länge 4.
  - ◇ `s.find("abc")`: Index des Teilstrings `"abc"`.

    Es wird der Index des ersten Vorkommens geliefert. Falls nicht gefunden, wird `string::npos` geliefert (z.B. -1).
  - ◇ `s.replace(2, 4, "abc")`: Ersetze Teilstring.

    Der Teilstring mit Index 2–5 (Länge 4) wird durch `"abc"` ersetzt.
  - ◇ `s.insert(2, "abc")`: Füge `"abc"` ab Position 2 ein.
  - ◇ `s.erase(2, 3)`: Lösche 3 Zeichen ab Index 2.
  - ◇ `s.c_str()`: Wert als C-String.

Inhalt

1. Einführung, Arraygrenzen-Verletzung
2. C-Strings, Klasse string
3. Initialisierung von Arrays
4. Mehrdimensionale Arrays
5. Zeiger (Pointer): Einführung
6. Zeiger und Arrays, Zeiger-Arithmetik

Array Initialisierung (1)

- Man kann Arrays mit einer Liste von Werten in geschweiften Klammern initialisieren:

```
int a[5] = { 27, 42, 18, 73, 56 };
```

- Falls man zu wenig Werte angibt, werden die übrigen Array-Elemente mit 0 gefüllt:

```
int a[5] = { 27, 42, 18 };
```

- Zu viele Werte anzugeben, ist dagegen ein Fehler.
- Zeichenarrays können mit einer String-Konstante initialisiert werden:

```
char a[5] = "abc";
```


Array Initialisierung (2)

- Wenn man eine Initialisierungsliste verwendet, kann der Compiler die Array-Größe selbst bestimmen:

```
int a[] = { 27, 42, 18, 73, 56 };
```

  Oft verwendet man in solchen Fällen 0 oder -1 als letztes Array-Element, um das Ende zu markieren. Übrigens darf man in C/C++ ein Komma auch nach dem letzten Element der Liste schreiben.

- Initialisierungen sind auch eine Art von Zuweisung.

  Im ursprünglichen C durften konsequenterweise nur solche Variablen initialisiert werden, bei denen die Initialisierung schon beim Compilieren erfolgen kann (globale und statische Variablen). Diese Einschränkung ist in C++ und ISO-C entfallen.

Inhalt

1. Einführung, Arraygrenzen-Verletzung
2. C-Strings, Klasse string
3. Initialisierung von Arrays
4. Mehrdimensionale Arrays
5. Zeiger (Pointer): Einführung
6. Zeiger und Arrays, Zeiger-Arithmetik

Mehrdimensionale Arrays (1)

- Der Element-Typ von einem Array kann selbst wieder ein Array sein.
- Z.B. deklariert man mit

```
int a[3][5];
```

  ein Array der Größe 3, dessen Elemente Arrays der Größe 5 von Integers sind.
- Beim Zugriff `a[i][j]` muß entsprechend der Wert von `i` zwischen 0 und 2 liegen, und der Wert von `j` zwischen 0 und 4.

Mehrdimensionale Arrays (2)

- Man kann sich das Array `a[3][5]` als 3×5-Matrix vorstellen (3 Zeilen, 5 Spalten).

  Eigentlich spielt es keine Rolle, was man für die Zeile und was man für die Spalte hält, solange man es nur konsequent handhabt. Für die Vorstellung, daß der erste Index die Zeile und der zweite die Spalte angibt, spricht aber die Syntax der Initialisierung und eventuell auch die übliche Darstellung des Speicher-Layouts.

- Die Initialisierung ist möglich mit

```
int a[3][5] = {
          {  1,  2,  3,  4,  5 },
          {  6,  7,  8,  9, 10 },
          { 11, 12, 13, 14, 15 }
};
```


Mehrdimensionale Arrays (3)

- Hat das Array `a[3][5]` die Startadresse s, und ist jedes Array-Element 4 Byte lang, so wird a$[i][j]$ an Adresse $s + (i * 5 + j) * 4$ gespeichert.
- Die hintere(n) Dimension(en) werden also notwendig für die Adressberechnung gebraucht, die erste nicht unbedingt.

  Für Prozedurparameter (s.u.) kann man z.B. die unvollständige Typ-Angabe `int a[][5]` verwenden.

- Bei der Initialisierung darf man die erste Dimension weglassen, aber nicht die folgenden:

```
int a[][5] = { ... };
```

Mehrdimensionale Arrays (4)

- Mehrdimensionale Arrays werden oft mit geschachtelten Schleifen verarbeitet, z.B. kann man alle Elemente von `a` so mit 0 initialisieren:

```
for(int i = 0; i < 3; i++)
        for(int j = 0; j < 5; j++)
                a[i][j] = 0;
```

- So werden alle Kombinationen von Werten für `i` und `j` betrachtet.

  Für jeden Wert von `i` werden alle möglichen Werte von `j` durchgegangen.

Mehrdimensionale Arrays (5)

Warnung:

- Aus der Mathematik ist die Schreibweise

  `a[i,j]`

  bekannt. Diese Schreibweise ist nicht korrekt in C++.

  Besonders ungünstig ist, dass `a[i,j]` gültiges C++ ist: Es wird allerdings so interpretiert, dass hier nur ein Index angegeben ist, dessen Wert mit dem Komma-Operator berechnet wird. Es bedeutet also das gleiche wie `a[j]`. Oft wird man einen Typ-Fehler bekommen, weil `a[i,j]` nicht den erwarteten Typ `int` hat, sondern den Typ `int*` (siehe unten). Aber wenn man es z.B. mit << ausgibt, wird die Adresse (in Hexadezimalschreibweise) gedruckt — es gibt keine Fehlermeldung. Stellt man dagegen Warnungen an, bekommt man: "left-hand operand of comma has no effect". In Pascal wäre `a[i,j]` übrigens korrekt, und einfach eine Abkürzung für `a[i][j]`.

Inhalt

1. Einführung, Arraygrenzen-Verletzung
2. C-Strings, Klasse string
3. Initialisierung von Arrays
4. Mehrdimensionale Arrays
5. Zeiger (Pointer): Einführung
6. Zeiger und Arrays, Zeiger-Arithmetik

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 $\mathit{Pointer}(\texttt{int})$ für den Typ von p.
- Mit dem Operator `&` kann man die Adresse einer Variablen bestimmen. Er hat die Signatur

  $$\mathit{Lvalue}(T) \ \rightarrow \ \mathit{Pointer}(T)$$

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

  $$\mathit{Pointer}(T) \ \rightarrow \ \mathit{Lvalue}(T)$$

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:

1000:	1	⟵	m
1004:	2	⟵	n
1008:	1004	⟵	p

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++`). Siehe Kapitel 10.

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);`

Inhalt

1. Einführung, Arraygrenzen-Verletzung
2. C-Strings, Klasse string
3. Initialisierung von Arrays
4. Mehrdimensionale Arrays
5. Zeiger (Pointer): Einführung
6. Zeiger und Arrays, Zeiger-Arithmetik

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

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

- 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

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

- 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:

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

  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

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

    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:

$$\mathit{Lvalue}(\mathit{Pointer}(T)) \;\rightarrow\; \mathit{Pointer}(T).$$

  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.