Programmierkurs C/C++

"Mittwoch"

Sandro Andreotti
andreott@inf.fu-berlin.de
WS 2010/11
(Slides: Andreas Döring)

Arrays variabler Länge?

... versuchen wir mal was:

```
int count;
scanf("%i", & count);
int a [count];
```

⇐ Fehler beim Compilieren

Die Dimensionsangaben bei der Definition von Arrays muss eine zur Compilezeit bekannte Konstante sein!

Der Gnu-C++-Compiler allerdings lässt den oben beschriebenen Code durchgehen. Das ist allerdings nicht Standardkonform und sollte deswegen vermieden werden.

new

Mit **new** kann man zur Laufzeit Speicher für Variablen reservieren.

Bsp.:

```
int count;
scanf("%i", & count);

int * a = new int[ count ];
```

Der Speicher wird auf einem besonderen Speicherbereich, dem „Heap“, reserviert.

new liefert immer einen Pointer auf den allokierten Speicherbereich zurück.

Zusätzlich zur Speicherung des Arrays (auf dem Heap) wird also noch eine Zeigervariable **a** vom Typ **int *** benötigt, in welcher die Startadresse des Arrays gespeichert wird.

Man kann auch einzelne Objekte auf dem Heap allokieren, z.B. so:

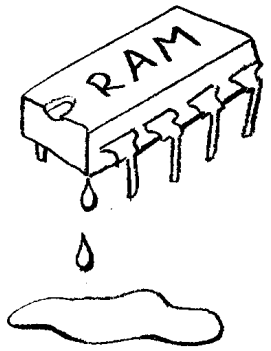
int * a = new int;

Zu jedem **new** gehört ein **delete**!

Ein mit **new** reservierter Speicherbereich muss irgendwann mit **delete** wieder freigegeben werden:

C++

```
char * a = new char [100]);
// ...
// verwende a
// ...
delete a;
```



memory leak

delete braucht als Argument einen Pointer, der exakt auf den Anfang des mit **new** reservierten Speicherbereiches zeigt.

Vergisst man, den Speicher wieder freizugeben, so kann der Speicher nicht wieder neu vergeben werden. Das kann über kurz oder lang dazu führen, dass kein Speicher mehr zur Verfügung steht.

Wichtig für später: Hat man mit **new** ein Array **a** von „ausgewachsenen“ Objekten allokiert, und möchte, dass beim **delete** die *Destruktoren* all dieser Objekte aufgerufen wird, so muss man statt dessen

delete [] a;

schreiben. Hier wird also nicht der Operator „**delete**“ sondern der operator „**delete[]**“ verwendet.

Speicher-Arten

Speicherarten für Variablen:

1. static
   - leben so lange das Programm läuft
   - globale Variablen (u.A.)
1. auto ⇒ Stack
   - leben bis { }-Block verlassen wird
   - lokale Variablen
1. dynamic ⇒ Heap
   - mit **`new`** erzeugt
   - leben bis zum **`delete`**

Andere Beispiele für static Variablen sind:

-statische lokale Variablen

-statische Member-Variablen

Schlüsselwort *static*

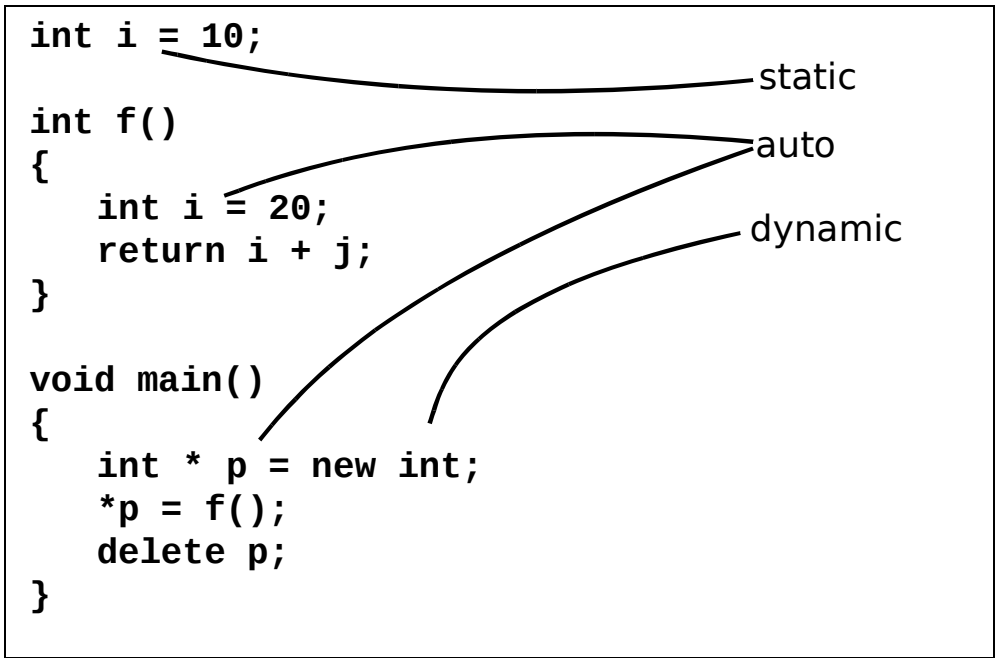
Mit static kann man eine lokale Variable von einer Auto- zu einer Static-Variablen machen.

Bsp.:

```
void ichZaehleMit()
{
   static int count = 0;
   ++count;
   printf("Aufruf %i, ", count);
}
```

Die Variable **counter** ist lokal, d.h. sie ist nur innerhalb der Funktion **ichZaehleMit()** bekannt, und sie ist statisch, d.h. sie wird nicht bei jedem Funktionsaufruf neu erzeugt und weggeworfen, sondern nur einmal beim ersten Aufruf der Funktion **ichZaehleMit()** angelegt und initialisiert. Spätere Aufrufe der Funktion verwenden immer die selbe Variable **count**. Die Variable „lebt“ bis zum Ende des Programmes.

storage duration : Beispiel



Man beachte: **p** selbst ist eine auto-Variable, jedoch die Variable, auf die **p** zeigt, liegt auf dem Heap und ist somit eine dynamic-Variable.

lokale Variablen sterben jung

```
int * getPrimArray()
{
   int arr[] = {2, 3, 5, 7};
   return arr;
}

int * prims = getPrimArray();
```

⇐ Vorsicht!

Variable **`arr`** wird nach Ende der Funktion **`getPrimArray()`** zerstört.

Das Problem kann dadurch behoben werden, dass man **arr** zu einer static-Variablen macht, wie im folgenden beschrieben wird. Besser wäre es aber vermutlich, **arr** einfach mit **new** zu erzeugen. In diesem Fall müsste der Benutzer von **getPrimArray** allerdings selbst darauf achten, nach Benutzung des Arrays dieses auch wieder mit **delete** freizugeben. Bestimmt wäre es deshalb noch besser, Der Funktion **getPrimArray** ein leeres Array zu übergeben, das dann von der Funktion mit Prinzahlen gefüllt wird.

Quiz: Was wird ausgegeben?

```
int p(char * str)
{
  printf("%s\n", str);
  return 0;
}

int global = p("global");

void f()
{
  p("f");
  static int local = p("local");
}

int main()
{
  p("start"); f(); f(); f(); p("stop");
  return 0;
}
```

Ausgabe:

global

start

f

local

f

f

stop

Stack ist schneller als Heap!

Regel: » Benutze den Heap nur wenn du musst! «

Gründe für Heap:

- Variable zu groß für den Stack
- Größe/Anzahl zur Compilezeit nicht bekannt
- Lebenszeit der Variable soll dynamisch kontrolliert werden

In allen anderen Fällen sind Stack-Variablen vermutlich die bessere Wahl, da der Aufruf von **free**/**delete** einen nicht unbeträchtlichen Overhead erzeugen kann. Außerdem birgt der Stack auch nicht die Gefahr von Memory Leaks.

Mehrdimensionale Arrays?

Wir wollen ein dynamisches, zweidimensionales Array erstellen:

```
int count;
scanf("%i", & count);
??? arr = new int[count][count];
```

Antwort: Es geht nicht!

Bei mehrdimensionalen Arrays bleibt einem nichts anderes übrig, als z.B. ein eindimensionales Array der richtigen Größe zu erzeugen, und darauf ein mehrdimensionales Array "von Hand" zu emulieren. In einer der heutigen Aufgaben haben Sie die Gelegenheit, etwas dieser Art auszuprobieren.

Die C Standard Library

Überblick: Die Header-Dateien für C

C Library Header Files

<assert.h>	<inttypes.h>	<signal.h>	<stdlib.h>
<complex.h>	<iso646.h>	<stdarg.h>	<string.h>
<ctype.h>	<limits.h>	<stdbool.h>	<tgmath.h>
<errno.h>	<locale.h>	<stddef.h>	<time.h>
<fenv.h>	<math.h>	<stdint.h>	<wchar.h>
<float.h>	<setjmp.h>	<stdio.h>	<wctype.h>

[Für eine Dokumentation siehe z.B.
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/]

<stdio.h>

- Formatierte Ein- und Ausgabe:
  printf, scanf, ...
- Dateifunktionen:
  fopen, fwrite, fclose, remove, rename, ...

```
FILE * stream;
if (stream = fopen("test.txt", "w"))
{
  fprintf(stream, "hallo");
  fclose(stream);
}
```

<string.h>

- Memory Manipulation:

  memset, memmove, memcmp, ...

```
char a[100];
char b[100];

memset(a, 'C', 100);
memmove(b, a, 100);
```

- String Manipulation:

  strcpy, strchr, strstr, strcmp, strlen...

```
int x = strlen("hallo"); //x = 5
```

Anmerkung: **strlen** auf ein String Literal (wie im Beispiel) anzuwenden, macht allerdings nicht viel Sinn, da man in diesem Fall nämlich auch **sizeof(„hallo“)-1** verwenden könnte. Das würde zur Compilezeit ausgewertet werden, während **strlen** Lauzeit kosten würde.

<math.h>

- Trigonometrische Funktionen:
  sin, cos, tan, asin, cosh, ...
- Exponentialfunktion und Logarithmus:
  exp, sqrt, log, log10, ...
- sonstige:
  ceil, floor, ...

<time.h>

- Funktionen für Uhrzeit und Datum:
  clock, localtime, ...

```
clock_t start = clock();

// ... das kostet Zeit

clock_t stop = clock();
printf("Zeit: %i ticks", stop – start);
```

Die Ticks pro Sekunde sind in **CLOCKS_PER_SECOND** definiert. Üblich sind 100 Ticks pro Sekunde.

Vorsicht: Zeitmessungen dieser Art sind gewöhnlich wenig genau (eben z.B. nur auf eine hundertstel Sekunde).

`<stdlib.h>`

Enthält alles Mögliche, u.A.:

- String-Konversionsfunktionen:
  `atoi, atof, strtol, ...`
- Pseudo-Zufalls-Funktionen:
  `rand, srand`
- Systemfunktionen:
  `exit, getenv, ...`

srand startet den Zufallsgenerator mit einem Saatwert („seed“), in diesem Fall mit **time(0)**, d.h. der aktuellen Uhrzeit. Dies braucht man nur einmal zu Beginn des Programmes zu tun. **rand()** erzeugt zufällige Zahlen zwischen **0** und **RAND_MAX**.

Die C++ Standard Library

Überblick: Die Header-Dateien für C++

C Library Header Files for C++

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<ctime>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwchar>
<cerrno>	<clocale>	<cstdarg>	<cstring>	<cwctype>
<cfloat>	<cmath>	<cstddef>		

C++ Library Header Files

<algorithm>	<iomanip>	<list>	<ostream>	<streambuf>
<bitset>	<ios>	<locale>	<queue>	<string>
<complex>	<iosfwd>	<map>	<set>	<typeinfo>
<deque>	<iostream>	<memory>	<sstream>	<utility>
<exception>	<istream>	<new>	<stack>	<valarray>
<fstream>	<iterator>	<numeric>	<stdexcept>	<vector>
<functional>	<limits>			

32 C++ Library Headers und 18 C Library Headers.

Die C-Library Header entsprechen einem Teil der Header, die auch bei C Standard sind, jedoch mit einem zusätzlichen „c“ am Anfang, und ohne „.h“ am Ende. Statt „stdio.h“ heißt der Header für die C standard Library zur Ein- und Ausgabe also „cstdio“. Die Definitionen in diesen Dateien sind (zum großen Teil) in den Namespace **std** gewandert. Ansonsten gibt es hier und da kleinere Unterschiede zu den üblichen C-Library Dateien.

[Literatur & Links:

- ISO-IEC- 14882 (1992): Programming languages - C++, Kapitel 17 - 27

-„C-Library Reference Guide“: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

]

Streams

Eine nettere Form der Ein- und Ausgabe:


```
#include <iostream>
using namespace std;
...
cout << "Zahl eingeben: ";
int i;
cin >> i;
cout << "Sie haben " << i << " eingegeben." << endl;
```

<algorithm>	<iomanip>	<list>	<ostream>	<streambuf>
<bitset>	<ios>	<locale>	<queue>	<string>
<complex>	<iosfwd>	<map>	<set>	<typeinfo>
<deque>	<iostream>	<memory>	<sstream>	<utility>
<exception>	<istream>	<new>	<stack>	<valarray>
<fstream>	<iterator>	<numeric>	<stdexcept>	<vector>
<functional>	<limits>			

In den meisten Fällen ist es ausreichend und das einfachste, **iostream** einzubindet.

Für file streams bindet man **fstream** ein.

Container

z.B. Strings:

```
#include <string>
using namespace std;
...
string s = "hallo ";
s += "Berlin";
cout << s.length(); //Ausgabe: 12
```

<algorithm>	<iomanip>	<list>	<ostream>	<streambuf>
<bitset>	<ios>	<locale>	<queue>	<string>
<complex>	<iosfwd>	<map>	<set>	<typeinfo>
<deque>	<iostream>	<memory>	<sstream>	<utility>
<exception>	<istream>	<new>	<stack>	<valarray>
<fstream>	<iterator>	<numeric>	<stdexcept>	<vector>
<functional>	<limits>			

bitset: Sequenz einer festen Anzahl von Bits

deque: Vektor, der an beiden Enden schnell verlängert oder verkürzt werden kann.

list: Liste

map: Menge von (key, value)-Paaren, in der man schnell nach keys suchen kann.

queue: „fifo“-Schlange

set: Menge

stack: „lifo“-Stapel

string: Zeichenkette

vector: Sequenz von Objekten.

Weitere Inhalte der Standard Libraries kann man in der Literatur nachschlagen.

Fehlershow (1)

Frage: Was bedeutet folgende Fehlermeldung?
Zeit: Compilezeit

"... undeclared identifier" (Visual C++)

"... undeclared (first use this function)" (GCC)

Es wurde ein Bezeichner (Variable, Funktion, Klasse, ...) benutzt, ohne dass er zuvor deklariert worden ist. Dies könnte z.B. daran liegen, dass man die Deklaration einfach vergessen hat, oder es hat sich irgendwo ein Tippfehler eingeschlichen. Es könnte auch sein, dass man vergessen hat, eine Header-Datei einzubinden, in der das Symbol deklariert wird. Man beachte auch, dass ein Bezeichner in jeder Translation Unit deklariert werden muss, in der man ihn verwenden will.

Fehlershow (2)

Frage: Was bedeutet folgende Fehlermeldung?
Zeit: Compilezeit

"unresolved external symbol" **(Visual C++)**

"undefined reference to ..." **(GCC)**

Diese Meldung kommt vom Linker: Es wurde eine Funktion (oder eine Variable) deklariert und benutzt, aber in keiner Translation Unit tatsächlich definiert. Vielleicht hat man die entsprechende .c-Datei nicht mit in das Projekt eingebunden. Oder dem Linker fehlt eine Library, eventuell muss ein zusätzlicher Librarypfad angegeben werden.

Fehlershow (3)

Frage: Was bedeutet folgende Fehlermeldung?
Zeit: Laufzeit

"Access violation writing location..." (Windows)

"Segmentation fault" (Linux)

(Die Fehlermeldung unter Windows erhält man z.B. wenn man unter Visual Studio im Debugmodus arbeitet. Ansonsten wird man mit dem berühmten Popup beglückt, und darf auf Wunsch eine Nachricht an Microsoft schicken.)

Es wurde auf einen Speicherbereich zugegriffen, auf den nicht zugegriffen werden darf. Es könnte z.B. sein, dass mit **new**/**delete** zu wenig Speicher allokiert worden ist, und man über die zulässige Grenze hinaus geschrieben hat. Eine beliebte Methode, diese Art Fehler zu erzeugen, besteht darin, die 0 am Ende eines Strings zu vergessen. Eine andere häufige Ursache für diese Art Fehler ist, dass man versucht hat, einen Nullpointer zu dereferenzieren

Viele C++-Programmierer wundern sich darüber, dass ihr Programm nur manchmal einen solchen Fehler wirft, in anderen Situationen (z.B. auf anderen Rechnern oder ohne Optimierung Kompiliert) jedoch scheinbar problemlos läuft. Der Grund dafür ist einfach: Man kann sich nicht darauf verlassen, dass tatsächlich ein Fehler geworfen wird, wenn man über die Grenzen seines Speichers hinwegschreibt. Hat man nämlich Glück, so liegt dort ein Speicherbereich, in den man zufällig schreiben kann, ohne dass dies negative Auswirkungen hat.

Dieser Fehler ist besonders schwer zu debuggen, weil er der Fehler oft an einer ganz anderen Stelle zutage tritt, als er verursacht worden ist. Hier empfiehlt sich der Einsatz spezieller Tools, wie z.B. „Valgrind“.

Fehlershow (4)

Frage: Was bedeutet folgende Fehlermeldung?
Zeit: Laufzeit

"stack overflow" **(Windows)**

"Segmentation fault" **(Linux)**

Die maximale Größe des Stacks wurde überschritten. Dies kann z.B. dadurch geschehen, dass man bei einer rekursiven Funktion die korrekte Abbruchbedingung vergessen hat: Die Funktion ruft sich immer wieder selbst auf, bis der Stack voll ist. Auch wenn eine Anwendung ansonsten fehlerfrei ist, kann es zu einem derartigen Fehler kommen, wenn massiv rekursiv programmiert wird, oder zu viele zu große Stackvariablen verwendet werden. In diesem Fall muss man entweder sein Programm anpassen, oder aber man ändert die Einstellungen für die Stackgröße (dies ist eine Linker-Option).

Aufgaben zum Mittwoch:

7. Aufgabe:

Die **Edit-Distanz** dist(A, B) zwischen zwei Strings A und B ist definiert als die minimale Anzahl von Änderungen, die man an einem der Strings vornehmen muss, um ihn in den anderen String zu verwandeln. Als „Änderung“ stehen dabei folgende Operationen zur Verfügung: 1. Ein einzelnes Zeichen an einer beliebigen Stelle in den String einfügen, 2. ein beliebiges einzelnes Zeichen aus dem String löschen, 3. ein beliebiges Zeichen des Strings in ein andere Zeichen verwandeln. Diese Edit-Distanz kann ziemlich effizient unter Einsatz von **dynamischem Programmieren** berechnet werden: Statt nur dist(A, B) zu ermitteln, berechnen wir die Edit-Distanz dist(A‘, B‘) aller Paare A‘ und B‘, wobei A‘ ein Präfix von A ist (d.h. A fängt mit A‘ an) und B‘ ein Präfix von B ist. Wenn A‘ genau i Zeichen lang ist und B‘ genau j Zeichen, dann speichern wie dist(A‘, B‘) in einer (zweidimensionalen) Matrix M an der Stelle M(i, j), d.h. in der i-ten Zeile und der j-ten Spalte. Im Einzelnen sieht der Algorithmus dann so aus:

Initialisierung: $M(0, 0) = 0$

$M(i, 0) = i$, für $i > 0$

$M(0, j) = j$, für $j > 0$

Rekursion: $M(i, j) = \min(M(i - 1, j - 1) + d(i, j),$

$M(i-1, j) + 1,$

$M(i, j-1) + 1)$ für $i > 0$ und $j > 0$

Dabei ist $d(i, j) = 0$, wenn das i-te Zeichen von A gleich dem j-ten Zeichen von B ist, sonst ist $d(i, j) = 1$.

Schreiben Sie ein Programm, das zwei Strings einliest und die Edit-Distanz der beiden Strings berechnet.

8. Aufgabe:

(Erweiterung der PRÜFUNGSAUFGABE)

In dieser Aufgabe sollen sie einen Container aus der C++ Standard Library kennenlernen. Nutzen Sie den map Container um diesmal die Häufigkeiten von Wörtern zu zählen.

PRÜFUNGSAUFGABE:

Schreiben Sie ein Programm, das eine Textdatei einliest, die verschiedenen Buchstaben zählt, und anschließend eine Statistik darüber ausgibt, welche Buchstaben wie oft im Text vorkommt. Geben sie zur besseren Visualisierung eine art Histogramm aus wie z.B.

Text: „aaaaaaalllllooo“

a: 7: |||||||

l: 5: |||||

o: 3: |||