Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

- Kontrollstrukturen
  - Verzweigungen: `if` ... `else`, `switch` ... `case`
  - Schleifen: `for(..)`, `while(...)`, `do ... while (...)`
- Funktionen
  - Parameter
  - Scope
  - Call-by-Value vs. Call-by-Refenrece
  - Rekursion
- Sonstiges
  - assert()
  - Kommentare

Obfuscated C

```
int v,i,j,k,l,s,a[99];
main()
{
for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<
s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j)
&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),v||
(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
}
```

This program is about as simple and as readable as possible.
To make things even more simple we used a very limited subset of C:

- No pre-processor statements
- Only one, harmless, 'for' statement
- No ifs
- No breaks
- No cases
- No functions
- No gotos
- No structures

In short, it contains no C language that might confuse the innocent reader. :-)

Duff's Device (1)

- Aufgabe: Optimiere kopieren von einem Array in ein anderes
- Loop-Unrolling:

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

- Zuweisung schnell und effizient
- Aber Overhead beim Schleifenkopf: ++ und der Vergleich sind teuer im Vergleich zur Zuweisung
- Leistungszuwachs durch Sequentielle Zuweisung:

```
b[0] = a[0];
b[1] = a[1];
   ...
b[n] = a[n];
```

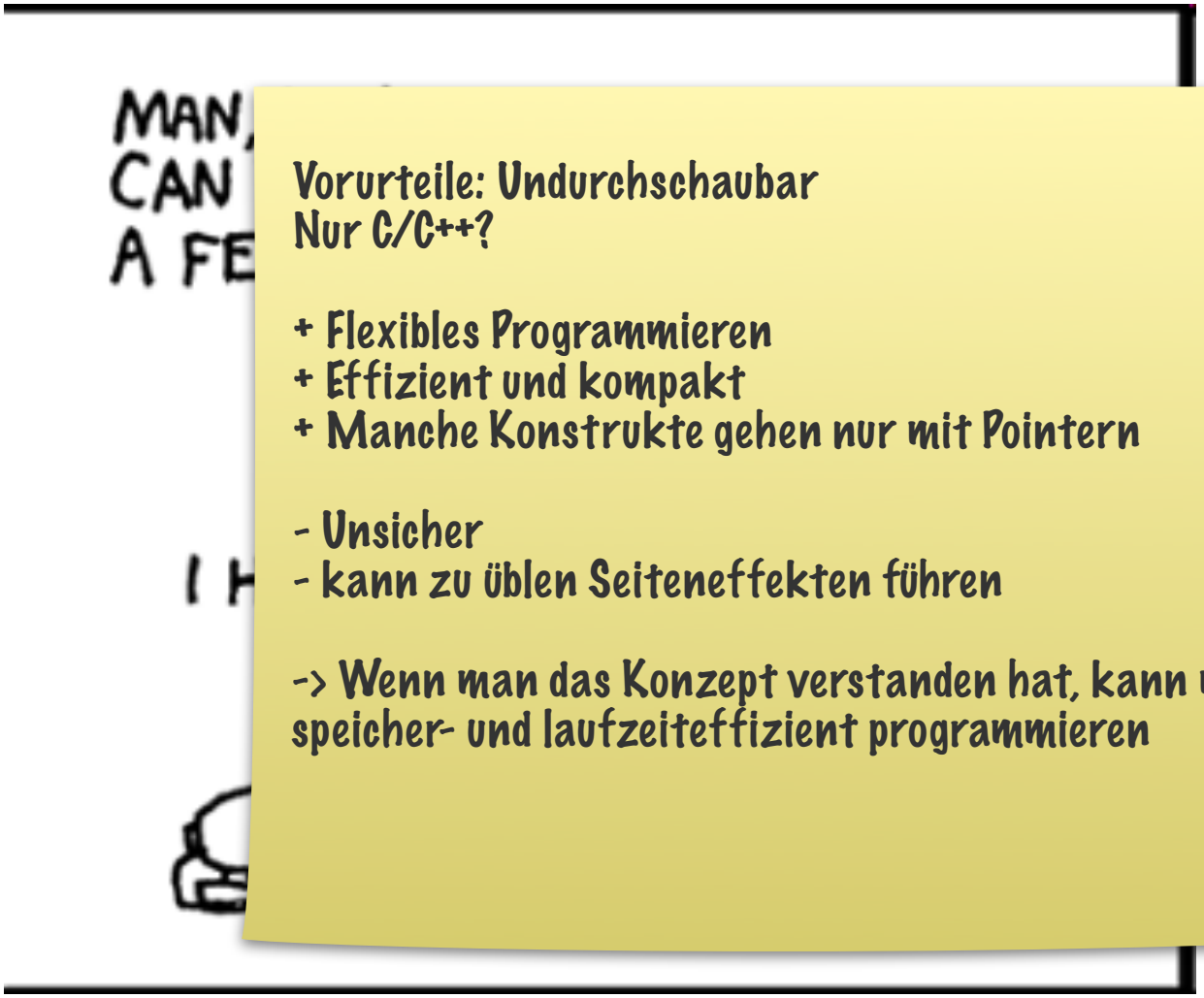
- Was ist wenn n variabel ist?
- Moderne Compiler können solche Fälle erkennen und machen das automatisch! Dennoch cool: Duff's Device

Duff's Device (2)

```
copy(int* to, int* from, int count)
{
   int n=(count+7)/8;
     switch(count%8){
     case 0:    do{ *to = *from++;
     case 7:        *to = *from++;
     case 6:        *to = *from++;
     case 5:        *to = *from++;
     case 4:        *to = *from++;
     case 3:        *to = *from++;
     case 2:       *to = *from++;
     case 1:        *to = *from++;
                }while(--n>0);
     }
}
```

- „This code forms some sort of argument in that debate, but I'm not sure whether it's for or against."
- „Disgusting, no? But it compiles and runs just fine. I feel a combination of pride and revulsion at this discovery. If no one's thought of it before, I think I'll name it after myself.

Nächstes Thema: Zeiger / Pointers



Vorurteile: Undurchschaubar
Nur C/C++?

+ Flexibles Programmieren
+ Effizient und kompakt
+ Manche Konstrukte gehen nur mit Pointern

- Unsicher
- kann zu üblen Seiteneffekten führen

-> Wenn man das Konzept verstanden hat, kann man speicher- und laufzeiteffizient programmieren

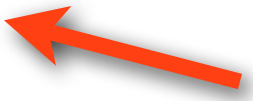
Gliederung

Zeiger / Pointer (1)

- Adresse
  - Ort im Speicher, an dem Daten gespeichert werden
  - Die Adresse einer Variablen erhält man durch den &-Operator
- Pointer
  - Eine typisierte Variable, die eine Adresse speichert
  - Wird durch * angezeigt

```
int  i = 10;
int *j = &i;          /* j "points" to i */
```

name	address	contends
i	0x123aa8	10
j	0x123aab	0x123aa8



Zeiger / Pointer (2)

- Der Inhalt (das Datum), auf den ein Pointer zeigt, ist mit dem `*`-Operator abrufbar:

```
int  i = 10;
int *j = &i;
printf("i = %d\n", i);
printf("j = %x\n", j);
printf("j points to: %d\n", *j);
```

- `&i` ist die Adresse von `i`
- `*j` ist der Inhalt des Speichers an der Stelle, auf den `j` zeigt
- `*`-Operator „dereferenziert“

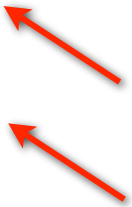
Bedeutungen von `*`:
1. Multiplikation
2. Deklaration eines Pointers
3. Dereferenzieren

Zeiger / Pointer (3)

- Jetzt geht‘s ab:

```
int   i = 10;
int  *j = &i;
int **k = &j;
printf("%x\t%d\n", &i, i);
printf("%x\t%x\t%d\n", &j, j, *j);
printf("%x\t%x\t%x\t%d\n", &k, k, *k, **k);
```

name	address	contends
i	0x123aa8	10
j	0x123aab	0x123aa8
k	0x123ab0	0x123aab



Zeiger / Pointer (4)

- Zuweisungen auf Pointer

```
int i = 10;
int *j = &i;
int *k;
/* Assign to contents of j: */
*j = 20;
/* Now i is 20.                    */
/* Assign j to k:                */
k = j;
/* Now k points to i too.    */
/* Assign to contents of j: */
*j = *k + i;
/* Now i is 40 */
```

- Wenn eine Pointer-Variable auf der linken Seite der Zuweisung steht, hängt das Programmverhalten davon ab, ob sie dereferenziert wird oder nicht

```
j = k;                    *j = *k + 10;
```

Zeiger / Pointer (5)

- Erinnerung: In C werden Variablen kopiert, bevor sie zu einer Funktion gesendet werden:

```
void incr(int i) {
    i++;
}
/* ... later ... */
int j = 10;
incr(j); /* want to increment j */
/* What is j now? */
/* Still 10 – incr() does nothing. */
```

  - **Call-By-Value**
  - Der Wert kann nur lokal in der Funktion geändert werden
- Oft will man aber den Wert einer Variablen verändern

  ➡ **Call-By-Reference**

Zeiger / Pointer (6)

- Call-By-Reference:

```
void incr(int *i) {
     (*i)++;
}
/* ... later ... */
int j = 10;
incr(&j);
/* What is j now? */
/* Yep, it's 11. */
```

- Alles klar?
- Das hatten wir zuvor bei `scanf()` und Co.:

```
int i;
scanf("%d", &i); /* read in i */
```

Zeiger / Pointer (7)

- Achtung:

```
void incr(int *i) {
    *i++;                /* Won't work!          */
                         /* Parsed as: *(i++); */
}
```

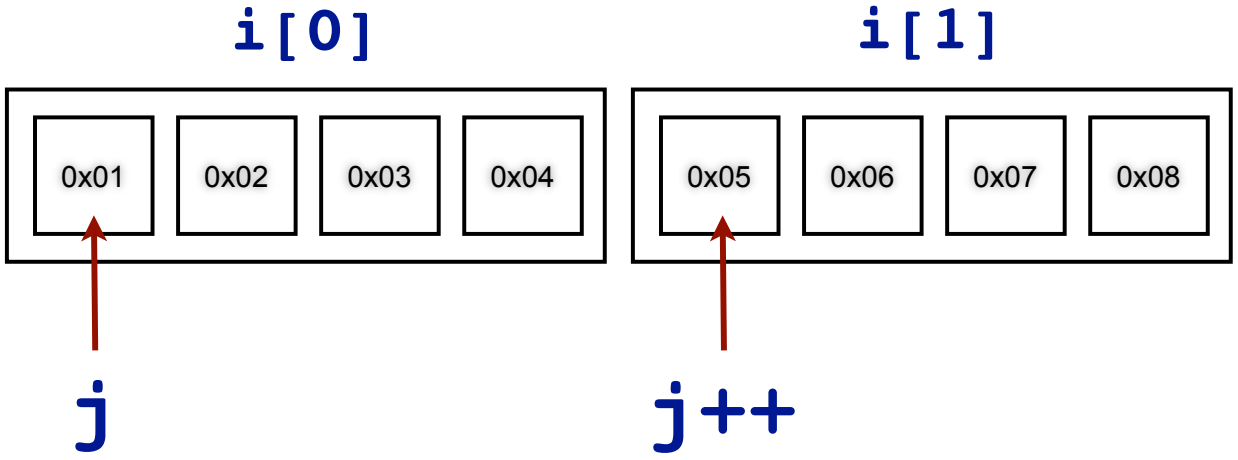
- Korrekt wäre: `(*i)++;`
- Benutzt `()` um Konfusion zu vermeiden!

Pointerarithmetik

- Man kann Integers von Pointern subtrahieren und addieren

```
int i[5] = { 1, 2, 3, 4, 5 };
int *j = i;   /* (*j) == ? */
j++;          /* (*j) == ? */
j += 2;       /* (*j) == ? */
j -= 3;       /* (*j) == ? */
```

- Pointerarithmetik addiert/subtrahiert Adressen nicht direkt, sondern verschiebt in Vielfachen der Bytezahl eines Typs:



Pointerarithmetik / Arrays und Pointer (1)

- Wie viele Bytes hat ein Integer? (vgl. Folie 25)

```
printf("size of integer: %d\n", sizeof(int));
printf("size of (int *): %d\n", sizeof(int *));
```

- Arrays sind versteckte Pointer!
- `i[3]` und `*(i+3)` sind identisch
- `i` ist identisch zu `&i[0]`

➡ **Pointerarithmetik statt Array-Operationen**

```
int array[1000];
for (i = 1; i < 998; i++) {
  array[i] = (array[i-1] + array[i] + array[i+1]) / 3;
}
```

```
int array[1000];
for (i = 1; i < 998; i++) {
  *(array+i) = (*(array+i-1) + *(array+i) + *(array+i+1)) / 3;
}
```

Pointerarithmetik / Arrays und Pointer (2)

- Beim Ausdruck `*(array + i)` muss eine Addition und eine Dereferenzierung ausgeführt werden
- Bei manchen Compilern geht Inkrementieren schneller!

```
int array[1000];
int *p1, *p2, *p3;
p1 = array; p2 = array+1; p3 = array+2;
for (i = 1; i < 998; i++) {
  *p2 = (*p1 + *p2 + *p3) / 3;
  p1++; p2++; p3++;
}
```

- Mehr Geschwindigkeitsgewinn bei 2D-Arrays!
- Übung: Wie kann das realisiert werden?

C's Eccentric View Of Arrays

- For C compilers `myArray[i]` is equivalent to `i[myArray]`.
- Experts know to put this to good use. To really disguise things, generate the index with a function:

```
int myfunc(int q, int p) { return p%q; }
...
myfunc(6291, 8)[Array];
```

Zeiger auf Funktionen: Function Pointer (1)

- Man kann auch Zeiger auf Funktionen vereinbaren:

```
int (*pt2Function)(int, int, int) = NULL;
```

- Zuweisung

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

int doMore(int a, int b; int c) {
      return a+b-c;
}

/* correct assigment using the address operator */
pt2Function = &doIt;

/* short form */
pt2Function = doIt;

/* switch pointer */
pt2Function = doMore;
```

Zeiger auf Funktionen: Function Pointer (2)

- Aufruf der Funktionen hinter den Zeigern:

```
int result = (*pt2Function) (12, 1, 2);
/* short version */
int result = pt2Function(12, 1, 2);
```

- Übergabe als Parameter:

```
void passPtr(int(*pt2Func)(int, int, int))
{
   int result = (*pt2Func) (12, 1, 2);
}

void pass_a_function_pointer()
{
   passPtr(&doIt);
}
```