Umgang mit Pointern

Hannes Röbe-Oltmanns

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg
Betreuer: Dr. Julian Kunkel

21.11.2013

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

informatik die zukunft

Gliederung

1 Einleitung

2 Typische Fehler im Umgang mit Pointern

3 Optimierung mit Pointern

4 Zusammenfassung

Was sind Pointer?

```
short myShort = MAX;

short* ptr = &myShort;
```

Adresse	Wert
....	......
0x00A8	0111 1111
0x00A9	1111 1111
0x00AA	0000 0000
0x00AB	0000 0100
......	......
0x00B1	0000 0000
0x00B2	0000 0000
0x00B3	1010 1000
0x00B4	0000 0000
0x00B5	0000 0000
0x00B6	0000 0000
0x00B7	0000 0000
0x00B8	0000 0000
....	......

Was sind Pointer?

```
short myShort = MAX;

short* ptr = &myShort;
```

Adresse	Wert
....	......
0x00A8	0111 1111
0x00A9	1111 1111
0x00AA	0000 0000
0x00AB	0000 0100
......	......
0x00B1	0000 0000
0x00B2	0000 0000
0x00B3	1010 1000
0x00B4	0000 0000
0x00B5	0000 0000
0x00B6	0000 0000
0x00B7	0000 0000
0x00B8	0000 0000
....	......

Was sind Pointer?

```
short myShort = MAX;

short* ptr = &myShort;

*ptr = 1;
```

Adresse	Wert
....	......
0x00A8	0111 1111
0x00A9	1111 1111
0x00AA	0000 0000
0x00AB	0000 0100
......	......
0x00B1	0000 0000
0x00B2	0000 0000
0x00B3	1010 1000
0x00B4	0000 0000
0x00B5	0000 0000
0x00B6	0000 0000
0x00B7	0000 0000
0x00B8	0000 0000
....	......

Was sind Pointer?

```
short myShort = MAX;

short* ptr = &myShort;

*ptr = 1;
```

Adresse	Wert
....	......
0x00A8	0000 0000
0x00A9	0000 0001
0x00AA	0000 0000
0x00AB	0000 0100
......	......
0x00B1	0000 0000
0x00B2	0000 0000
0x00B3	1010 1000
0x00B4	0000 0000
0x00B5	0000 0000
0x00B6	0000 0000
0x00B7	0000 0000
0x00B8	0000 0000
....	......

Was ist Pointer Arithmetik?

```
short* ptr = 0x1CA8;

short* ptr2 = ptr + 5;

ptr2 - ptr // = 5
```

Adresse	Wert
....	......
0x1CA8	0000 0000
0x1CA9	0000 0000
0x1CAA	0000 0000
0x1CAB	0000 0100
0x1CAC	0000 0000
0x1CAD	0000 0000
0x1CAE	0000 0000
0x1CAF	0000 0000
0x1CB0	0000 0000
0x1CB1	0000 0000
0x1CB2	0000 0000
0x1CB3	0000 0000
0x1CB4	0000 0000
0x1CB5	0000 0000

Was ist Pointer Arithmetik?

```
short* ptr = 0x1CA8;

short* ptr2 = ptr + 5;

ptr2 - ptr // = 5
```

Adresse	Wert
....	......
0x1CA8	0000 0000
0x1CA9	0000 0000
0x1CAA	0000 0000
0x1CAB	0000 0100
0x1CAC	0000 0000
0x1CAD	0000 0000
0x1CAE	0000 0000
0x1CAF	0000 0000
0x1CB0	0000 0000
0x1CB1	0000 0000
0x1CB2	0000 0000
0x1CB3	0000 0000
0x1CB4	0000 0000
0x1CB5	0000 0000

Was ist Pointer Arithmetik?

```
short* ptr = 0x1CA8;

short* ptr2 = ptr + 5;

ptr2 - ptr // = 5
```

Adresse	Wert
....	......
0x1CA8	0000 0000
0x1CA9	0000 0000
0x1CAA	0000 0000
0x1CAB	0000 0100
0x1CAC	0000 0000
0x1CAD	0000 0000
0x1CAE	0000 0000
0x1CAF	0000 0000
0x1CB0	0000 0000
0x1CB1	0000 0000
0x1CB2	0000 0000
0x1CB3	0000 0000
0x1CB4	0000 0000
0x1CB5	0000 0000

Pointer auf Strukturen

```
struct myStruct
{
    float X;
    float Y;
};

struct myStruct* ptr = &someStruct;

// Zugriff durch
(*ptr).X = 5;

// oder
ptr->X = 5;
```

Void Pointer

```
void* vptr;
```

- Pointer ohne spezifischen Typ
- Kann auf jeden beliebigen Typen zeigen
- Kann nicht dereferenziert werden
- Pointer Arithmetik nicht erlaubt

NULL Pointer

- Signalisiert, dass ein Pointer gerade auf kein Objekt im Speicher verweist
- Dereferenzierung führt zu "Undefined Behaviour"
  - In der Regel wird ein "Segmentation Fault" auftreten
- C-Standard erlaubt:

```c
#define NULL 0

// oder:
#define NULL (void*)0

//Verwendung:
int* ptr = NULL;
```

Wozu braucht man Pointer?

- Dynamische Reservierung von Speicher zur Laufzeit
  - Zu reservierende Speichergröße unbekannt
  - Zu groß für den Stack

```
int* myData = malloc(sizeof(int) * 1000);

// Zugriff auf einzelne Elemente:
*(myData + 5) = 100;

// oder:
myData[5] = 100;
```

Wozu braucht man Pointer?

- Um in Funktionen nicht auf Kopien arbeiten zu müssen
- Effizientere Übergabe von Objekten an Funktionen

```
int callByValue(int x, int y)
{
    // Mit Kopien arbeiten
    return x + y;
}

void callByReference(struct GraphNode* node)
{
    // node direkt bearbeiten
    node->Data = ...
}
```

Typische Fehler im Umgang mit Pointern

”Pointer sind die effektivste Art, sich in C in den Fuß zu schießen“[Dem]

Pointer Initialisierung

```
int* ptrOnBssSegment;

void myfunction()
{
    int* ptrOnStack;

    static int* ptrAlsoOnBss;

    int** ptrToPtrOnHeap = malloc(sizeof(int*));
    int* ptrOnHeap = *PtrToPtrOnHeap;
}
```

Pointer Initialisierung

```
int* ptrOnBssSegment;
// referenziert: NULL

void myfunction()
{
    int* ptrOnStack;
    // referenziert bspw.: 0x1f4000001f4

    static int* ptrAlsoOnBss;
    // referenziert: NULL

    int** ptrToPtrOnHeap = malloc(sizeof(int*));
    int* ptrOnHeap = *PtrToPtrOnHeap;
    // referenziert bspw.: 0x1f4
}
```

Wild Pointer

- Zeigt auf einen willkürlichen Speicherbereich
- Entsteht durch nicht initialisierte Pointer
- Führt bei Dereferenzierung zu "Undefined Behaviour"
- Behebung: Pointer bei Deklaration immer initialisieren:

```
int myInteger = 5;
int* ptr = &myInteger;

//oder:

int* ptr = NULL;
```

Allozierungsfehler

- Funktionen zur Speicherallozierung der stdlib können bei Fehler den Null Pointer zurückgeben
- Erkennen von Allozierungsfehlern durch Neuimplementation der Funktionen:

```
void* awareMalloc(size_t size)
{
    void* ptr = malloc(size);
    if(!ptr)
    {
      // Fehlermeldung ausgeben
      // Programm kontrolliert abbrechen
    }
    else
      return ptr;
}
```

Typsicherheit

- Pointer in C generell nicht typsicher
- Warnungen vom Compiler
- Einige nicht typsichere Funktionen:
  - malloc(size_t)
  - printf(const char* , ...)
  - memcpy(void*, const void*, size_t)

```
float myFloat = 5.5f;
float* floatptr = &myFloat;
int* intptr = floatptr;
```

Aliasing

- Alias: Zwei oder mehrere Pointer zeigen auf dieselbe Adresse
  - Fehler im Programm sind schwerer nachzuvollziehen
  - Gegebenenfalls unübersichtlicher Quellcode
  - Unbeabsichtigte Veränderung des referenzierten Bereichs
- Potentielle Aliase verhindern Compileroptimierungen

Strict Aliasing Rule

- Compilerannahme: Zwei Pointer unterschiedlichen Typs bilden keinen Alias
- Außnahmen:
  - char*
  - Structs zu ihren enthaltenen Datentypen
  - unsigned/signed/volatile/const Varianten

```
int anint;
void foo(double *dblptr)
{
   anint=1;
   *dblptr=3.14159;
   bar(anint);
}
// Aus Understanding C/C++ Strict Aliasing
```

Strict Aliasing Rule

- Compilerannahme: Zwei Pointer unterschiedlichen Typs bilden keinen Alias
- Außnahmen:
  - char*
  - Structs zu ihren enthaltenen Datentypen
  - unsigned/signed/volatile/const Varianten

```
int anint;
void foo(double *dblptr) //foo((double*)&anint)
{
   anint=1;
   *dblptr=3.14159;
   bar(anint);
}
// Aus Understanding C/C++ Strict Aliasing
```

Dangling Pointer

- Pointer, der auf nicht mehr reservierten Speicher zugreift
- Dereferenzierung führt zu "Undefined Behaviour"

```
int* ptr = malloc(sizeof(int));


free(ptr);

*ptr = 5; // Undefined Behaviour
```

Dangling Pointer

- Pointer, der auf nicht mehr reservierten Speicher zugreift
- Dereferenzierung führt zu "Undefined Behaviour"

```
int* ptr = malloc(sizeof(int));
int* ptrAlias = ptr;

free(ptr);

*ptrAlias = 6; // Undefined Behaviour
```

Dangling Pointer

- Behebung des Problems nur eingeschränkt möglich:

```
void safeFree(void **ptrToPtr)
{
   if(ptrToPtr != NULL)
   {
      free(*ptrToPtr);
      *ptrToPtr = NULL;
  }
}
```

- Ein Pointer-Alias von ptr kann immer noch zugreifen

Dangling Pointer

- Eine andere Möglichkeit einen Dangling Pointer zu erzeugen:

```
int* ptr = NULL;

void danglingPointer()
{
    int integerOnStack = 5;
    ptr = &integerOnStack;
}

*ptr = 5;
```

Valgrind

- Toolsammlung zum Debuggen von Programmen
- Bietet Unterstützung bei der Suche nach Speicherfehlern
  - Zeigt Zugriff auf nicht reservierten Speicher an
  - Kann fehlende Freigabe von Heapspeicher feststellen
  - Kann Cache Misses anzeigen
- Anleitung zur Benutzung auf der Seite des Fachbereichs
  - http://wr.informatik.uni-hamburg.de/teaching/ressourcen/debugging

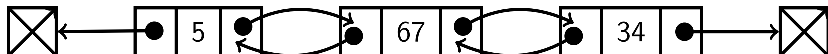
Optimierung mit Pointern

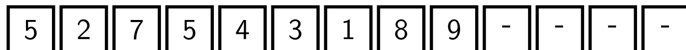
Folgende Problemstellung:

- Millionen von Elementen sollen verwaltet werden
- Elemente: struct Point{int X; int Y;};
- Häufigste Operationen: Entfernen/Einfügen
- Entfernen/Einfügen erfordert Suche

Welche Datenstruktur sollten wir verwenden?
-Eine doppelt verkettete Liste (Linkedlist)
-Ein dynamisch wachsendes Array (Arraylist)

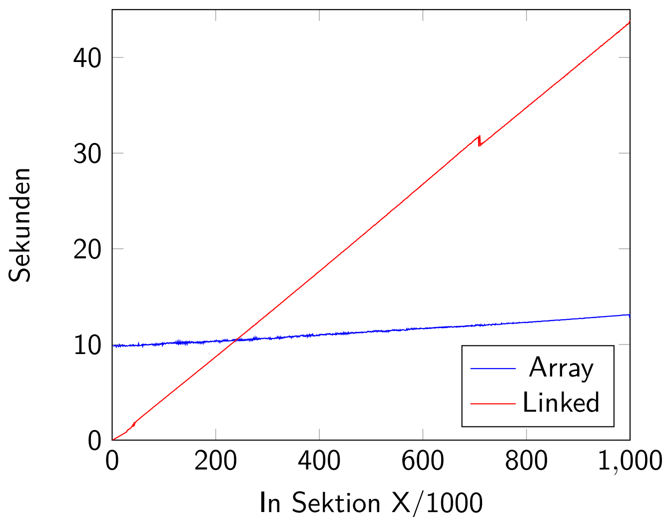
Listen





Linkedlist vs. Arraylist

- 10.000.000 Elemente
- 1000 Entfernungen
- Aufgeteilt in 1000 Sektionen



Call By Reference vs. Call By Value

```
typedef struct bigstruct
{
   char data[1000000];

}bigstruct;

void call_by_value(bigstruct big)
{
   // Mache irgendwas mit big
}

void call_by_reference(bigstruct* bigptr)
{
   // Mache irgendwas mit bigptr
}
```

Call By Reference vs. Call By Value

- 10.000-facher Aufruf der Funktionen ergibt:
  - Call By Value: 5,73 Sekunden
  - Call By Reference: 0,0 Sekunden
  - Call By Value musste ca. 10GB kopieren
  - Call By Reference musste ca. 80kB kopieren
- Performanz Gewinn sehr stark von der Größe des zu kopierenden Datentyps abhängig
- Call By Reference nicht generell besser als Call By Value

Umzeigen vs. Kopieren

- Analog zum vorigen Beispiel möchten wir große Kopiervorgänge vermeiden
- Anstelle Elemente innerhalb eines Arrays zu kopieren, werden Zeiger umgesetzt

Fazit zur Optimierung

- Pointer bieten viele Möglichkeiten zur Optimierung
- Aber: Führen häufig zu weniger lesbarem Code
- Grad der Optimierung apriori meist nicht bestimmbar
- Verkettete Datenstrukturen mit Pointern schlecht für effektive Cache Nutzung

Zusammenfassung

- Was Pointer sind
  - Pointer Arithmethik
  - Null Pointer
  - Void Pointer
  - Einsatzgebiete
- Pointer bereiten Probleme bei falscher Verwendung
  - Dangling/Wild Pointer
  - Allozierungsfehler
  - Alias Probleme
  - Typsicherheit
- Performanz Aspekte
  - Linkedlist vs. Arraylist
  - Call By Reference vs. Call By Value
  - Kopieren vs. Umzeigen

Ratschläge für den Umgang mit Pointern

- Pointer bei Deklaration immer direkt initialisieren
- Pointer nach free-Aufruf auf NULL setzen
- Mögliche Allozierungsfehler berücksichtigen
- Besondere Vorsicht bei Pointer Aliasing

Quellen I

Dennis M. Ritchie Brian W. Kernighan.
The C Programming Language, 1988.

Anthony Del Ciotto.
A Safe Free Memory Function in C.
http://anthdeldeveloper.wordpress.com/2013/09/22/a-safe-free-memory-function-in-c/.
[Online; letzter Zugriff 17.11.13].

Dr. Markus Demleitner.
http://www.cl.uni-heidelberg.de/kurs/skripte/prog2/html/page027.html.
[Online; letzter Zugriff 17.11.13].

Quellen II

Patrick Horgan.
Understanding c/c++ strict aliasing.
http://dbp-consulting.com/StrictAliasing.pdf.
[Online; letzter Zugriff 17.11.13].

Under what circumstances can malloc return null.
http://stackoverflow.com/questions/9101597/under-what-circumstances-can-malloc-return-null.
[Online; letzter Zugriff 17.11.13].

Pointer arithmetic for void pointer in c.
http://stackoverflow.com/questions/3523145/pointer-arithmetic-for-void-pointer-in-c.
[Online; letzter Zugriff 17.11.13].

Quellen III

Pointer.
http://en.wikipedia.org/wiki/Pointer_(computer_programming).
[Online; letzter Zugriff 17.11.13].

Pointer Aliasing.
http://en.wikipedia.org/wiki/Pointer_aliasing.
[Online; letzter Zugriff 17.11.13].

Dangling Pointer.
http://en.wikipedia.org/wiki/Dangling_pointer.
[Online; letzter Zugriff 17.11.13].

Quellen IV

C99-Standard.
http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, 2007.
[Online; letzter Zugriff 17.11.13].

Should we check if memory allocation fails.
http://stackoverflow.com/questions/7940279/should-we-check-if-memory-allocations-fail.
[Online; letzter Zugriff 17.11.13].

Bjarne Stroustrup.
Why You Should Avoid Linkedlists.
http://www.youtube.com/watch?v=YQs6IC-vgmol.
[Online; letzter Zugriff 17.11.13].