Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

13. September 2019

Pointer und Referenzen

Inhalt

- Pointer und Operatoren
- Wert-Zugriff über Pointer
- Pointer-Eigenschaften
- Pointer-Zuweisung
- `const`-Pointer
- `void`-Pointer
- Referenzen
- Memory Konzept
- Lebensdauer
- Return Typen und Lebensdauer

Pointer und Operatoren

- Pointer zeigt auf Memory im (virtuellen) Arbeitsspeicher
- Speicherbedarf entspricht Plattform: x86, x64, etc.
- Pointer wissen auf welchen Typ sie zeigen
- Mittels Adressoperator & kann von allem die Laufzeit-Adresse abgefragt werden
  - Variable, Funktion, Objekt, Methode
- Mittels Dereferenzierungsoperator * kann auf den gezeigten Typ zugegriffen werden
- Eine Referenz in Java entspricht ungefähr einem Pointer in C++

```
1  int x;          // nicht initialisierter Integer
2  int* px = &x;   // initialisierter Pointer auf Integer x
3  *px = 7;        // Wert von x wird auf 7 gesetzt
```

Wert-Zugriff über Pointer

Dereferenzierungsoperator * um auf gezeigte Werte zuzugreifen

```
1  int x;
2  int* px = &x;
3  *px = 3;                  // weisst x Wert '3' zu
4
5  MyClass y;
6  MyClass* py = &y;
7  auto z = *py;             // kopiert Instanz zu neuer z
8
9  int a = (*py).age;        // Zugriff auf Member 'age'
```

Wert-Zugriff über Pointer

Pfeiloperator -> um direkt auf Member zuzugreifen

```
1  struct MyContainer {
2    int values[];
3    void Sort();
4  };
5
6  MyContainer cont;
7  auto* p = &cont;
8
9  p->Sort();                  // Methoden-Aufruf
10 int* access = p->values;    // Wert kopieren
11 int value0 = p->values[0]; // Member direkt verwenden
```

Wert-Zugriff über Pointer

Indexoperator [] um auf Arrays zuzugreifen

```
1  char text[] = "Hello World";
2
3  char* pt1 = text;
4  char* pt2 = &text[0];
5  // pt2 zeigt auf gleiches Memory wie pt1
6
7  char h = pt1[0];
8  char w = pt2[6];
```

Pointer-Beispiele

```
1  char text[] = "abcd";    // text ist ein Pointer auf das
2                           // 'a' bzw. auf den ganzen
3                           // 0-terminierten C-String
4  char c = text[3];        // c enthält 'd'
5
6  char* p = text;          // p zeigt auf 'a'
7  char* x = text + 1;      // x zeigt auf 'b'
8  char* r = &x[1];         // r zeigt auf 'c'
9  char* s = &c;            // s zeigt auf 'd' in c
10 char* t = nullptr;       // oder in C: NULL
11
12 using PUInt = unsigned int*;
13 unsigned int val = 3;
14 PUInt pval = &val;       // pval zeigt auf 3 in val
```

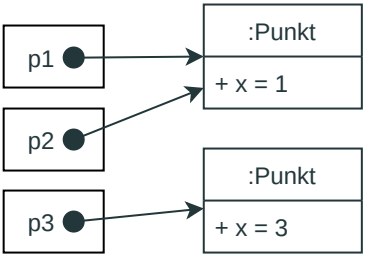
Pointer-Eigenschaften

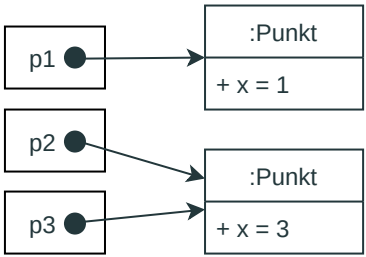
- Zeigen auf gültiges Memory, z.B:
  - Klassen-Instanzen, primitive Typen
  - das erste Element in einem Array oder C-String
- Zeigen auf ungültiges Memory, z.B:
  - `nullptr` (markiert als ungültiger Pointer)
  - uninitialisiertes Memory (Segfault, undefined behavior)
- Memory befindet sich im (virtuellen) Adressraum
  - Stack
  - Heap
  - gemapte Geräte
- Kennen Typ der "gezeigten" Instanz
  - Up-Cast immer möglich
  - Down-Cast mittels `dynamic_cast`

Pointer-Zuweisungen

```
1  Punkt *p1 = new Punkt();     // zeigt auf Objekt auf Heap
2  Punkt *p2 = nullptr;
3  Punkt *p3 = new Punkt();
4  p2 = p1;          // p2 zeigt auf gleiches Objekt wie p1
5  p2 = p3;          // p2 zeigt auf gleiches Objekt wie p3
```

Achtung: die Punktdaten werden nicht kopiert





const-Pointer

- const-Qualifier kann zwei Dinge bedeuten:
  - Adresse ist unveränderbar (Wert des Pointers)
  - Wert der gezeigten Instanz ist unveränderbar
- const beeinflusst das was links ist (ausser wenn nichts links)

```
1  int x;
2  // beide veränderbar
3  int* p = &x;
4  // Wert unveränderbar, Pointer veränderbar
5  const int* cp = &x;          // oder: int const *
6  // Wert veränderbar, Pointer unveränderbar
7  int* const pc = &x;
8  // beide unveränderbar
9  const int* const cpc = &x;  // oder: int const * const
```

`void`-Pointer

- ermöglicht Pointer ohne Typ → kein Zugriff
- manueller Typ-Cast nötig
- sollte nur in C verwendet werden
- meist für Pointer auf rohen Speicher

```
1 void* memset(void* dest, int byte_value, size_t count);
```

- verhält sich ähnlich der `Object`-Basis-Klasse in Java

```
1 // impliziter up-cast
2 void* a = new MyClass();
3 // expliziter down-cast
4 MyClass* b = dynamic_cast<MyClass*>(a);
```

Referenzen

- Alias für eine andere Variable (lvalue reference)
- Anstatt * wird hier & beim Typ verwendet
- muss immer initialisiert sein und ist unveränderbar
- verhindern Kopien z.B. bei Parameter-Übergabe

```
1  int x;
2  int& rx = x;     // rx ist ein Alias für x
3  rx = 3;          // Wert '3' an x zuweisen
```

Referenzen sind "immateriell"

- werden meist wegoptimiert
- haben deshalb keine Repräsentanz im Memory

Referenzen vs. Pointer

- Referenzen brauchen keine Dereferenzierungs- oder Adress-Operatoren
- Referenzen sind immateriell
- wenn nicht werden sie Compiler intern als Pointer implementiert
- Referenzen sind unveränderbar
- referenzierter Wert kann veränderbar sein

Referenzen und Pointer

- Referenzen "speichern" auch nur Adressen auf Variablen
- Adressoperator & verwenden um diese Adresse zu erhalten

```
1  int x;
2  int& rx = x;
3  int* px = &x;        // px erhält Adresse von x
4  px = &rx;            // dito
```

- Referenzen auf Pointer und umgekehrt erlaubt

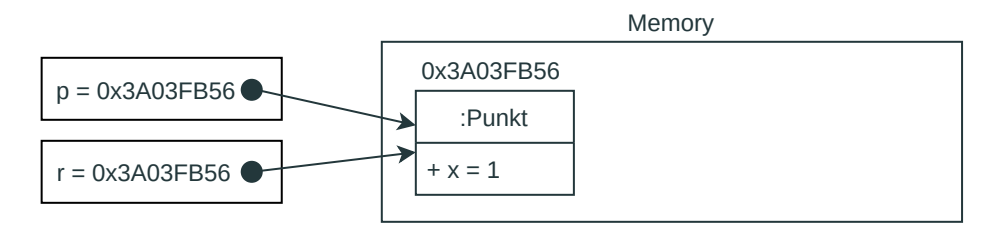
```
1  int x;
2  int* px = &x;
3  int*& rpx = px;
4  *rpx = 3;            // Wert '3' an x zuweisen
```

Memory Konzept

- Pointer zeigen immer auf rohes Memory
- Typ des Pointers erlaubt Zugriff auf abstraktere Konzepte

```
1 Punkt* p;
2 // entspricht
3 std::uintptr_t r;   // unsigned Pointer Typ
```

- `r` weiss aber nichts von `Punkt`
- `std::uintptr_t` ein expliziterer Alias für `void*`



Heap-Memory

C (Funktionen)	C++ (Operatoren)
`malloc`, `calloc`, `realloc`	`new`, `new ...[]`
`free`	`delete`, `delete[]`

```
1 auto* p = new Punkt();
2 assert(p != nullptr);
3 delete p;
```

```
1 auto* pa = new Punkt[4];
2 assert(pa != nullptr);
3 delete[] pa;
```

Nicht nur Memory-Allocator!

- new: Alloziert Memory und ruft ctor auf.
- delete: Ruft dtor auf und gibt Memory frei.

Lebensdauer: Stack

- statische Variablen haben static lifetime
- lokale Variablen haben automatic lifetime
- “Lokalität” wird durch Scopes definiert
- hat prinzipiel nichts mit Heap oder Stack zu tun

```
1  int main() {
2    Punkt* py = nullptr;
3    {
4      Punkt y;              // lokale Variable y
5      py = &y;
6    }                       // dtor von y
7    auto value = *py;       // Zugriff auf y -> segfault!
8  }
```

Lebensdauer: Heap

```
1  int main() {
2    Punkt* py = nullptr;
3    {
4      auto* y = new Punkt();  // lokale Variable y
5      py = y;
6    }                         // kein dtor von y!
7    auto value = *py;         // Zugriff auf y
8    delete py;                // dtor von y
9  }
```

- `delete` begrenzt die Lebensdauer von Heap-Instanzen, nicht Lokalität
- Heap-Instanzen sollten immer in Klassen mit dtor gekapselt sein

Return Typen und Lebensdauer

```
1  int& CreateInt() {
2    int a = 11;
3    return a;
4  }           // a wird zerstört
5
6  int& b = CreateInt();
7  int c = CreateInt();
```

Wie gebe ich Werte zurück?

1. Kopie zurückgeben
2. als Smart-Pointer
3. als Out-Parameter
4. als const-reference (nur in speziellen Performance-relevanten use-cases)