Programmieren in C++

Zeiger, Referenzen und Klassen

Inhalt

- Zeiger
- Konstante Zeiger
- Referenzen
- Klassen
- Neue Zeigerobjekte (C++11)
- Smart Pointers (C++11)

Zeiger und Adressoperator

- Infos
  - ein Zeiger zeigt auf eine Speicherstelle des (virtuellen) Adressraums
  - Zeiger werden auf 32-Bit-Plattformen mit 32 Bit abgespeichert
  - Zeigen können im Quellcode Typinformationen mitführen
  - von jeder Variable und jedem Objekt kann mit dem Adressoperator & zur Laufzeit die Adresse (Speicherstelle) abgefragt werden
- Beispiele

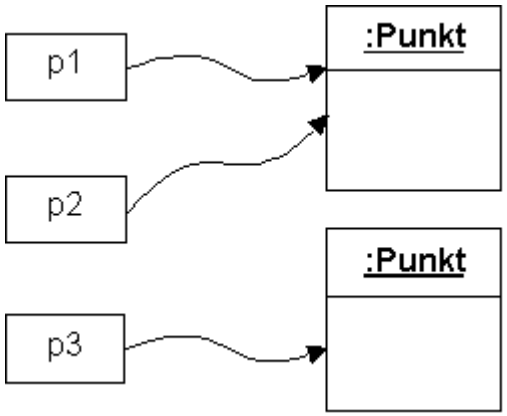
```
typedef unsigned int * PUInt32;
char text[] = "test";
unsigned int i = 2;
char c = text[i + 1];
char *p = text, *q = text + 1, *r = &text[i], *s = &c, *t = nullptr, *u = 0;
PUInt32 x = &i;
void *y = x;
```

Zeigervariablen bzw. Zeiger

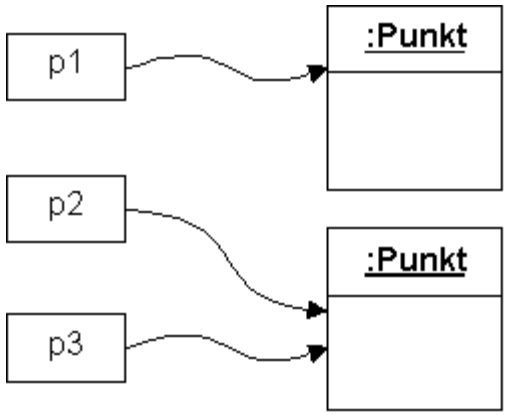
- zeigen auf gültige Speicheradressen, z.B.
  - dynamische Objekte
  - aufs erste Element von Arrays
  - auf statische Variablen und Objekte (Achtung Lebensdauer!)
- zeigen auf ungültige Speicheradressen
  - nullptr (0 oder NULL)
  - uninitialisierter Speicherbereich
- haben einen Typ „Zeiger auf …“
  - soll eine Zeigervariable auf eine Instanz der Klasse C zeigen, so muss der Typ der Zeigervariablen zur Klasse C zuweisungskompatibel sein
  - Beispiele: Punkt *p1;
    ~~p1 = new Person();~~

Zuweisungen bei Zeigervariablen

```
Punkt *p3 = new Punkt();
Punkt *p2, *p1 = new Punkt();
p2 = p1;
```




```
p2 = p3;
```



void-Zeiger

- Schlüsselwort void
  - wie in Java: leerer Rückgabetyp zur Unterscheidung von Prozeduren und Funktionen
  - nur in C++
    - leere Parameterliste einer Methode (optional)
    - void-Zeiger
- void-Zeiger
  - zeigt auf untypisierten Speicher
  - sind einseitig zuweisungskompatibel mit allen anderen Zeigertypen
    - jeder beliebige Zeiger kann an einen void* zugewiesen werden
    - ein void* kann nicht ohne Typenkonvertierung einem andersartigen Zeiger zugewiesen werden
  - typischer Einsatz im Low-Level-Bereich im Umgang mit Datenpuffern
  - Einsatz wenn möglich vermeiden

Zeiger und Konstanten

- Schlüsselwort const in Verbindung mit Zeigervariablen
  - was soll ausgedrückt werden?
    - Wert der Zeigervariablen ist unveränderbar (Adresse ist konstant)
    - Wert der Variablen, auf die der Zeiger zeigt, ist unveränderbar
- 4 Variationen

  int x;
  - nichts ist konstant
    - int *p = &x;
  - nur Ziel ist konstant: p ist ein Zeiger auf einen konstanten Integer
    - const int *p = &x;
  - nur Zeiger ist konstant: p ist ein konstanter Zeiger auf einen Integer
    - int *const p = &x;
  - Ziel und Zeiger sind beide konstant
    - const int *const p = &x;

Referenzen (nur in C++, nicht in C)

- Fakten
  - eine Referenz ist ein Alias für eine andere Variable (sog. lvalue)
  - eine Referenz wird durch ein & gekennzeichnet
  - eine Referenz kann nicht uninitialisiert sein
  - eine Neuinitialisierung ist nicht möglich
  - hinter der Kulisse ist eine Referenz nichts Anderes als ein Zeiger

- Beispiele

```
int k = 2;
int& ref = k;          // ref ist ein Alias für k
ref = 3;               // die Variable k hat nun den Wert 3
int *pk = &k;
int*& ref2 = pk;       // ref2 ist ein Alias für den Zeiger pk
*ref2 = 4;             // die Variable k hat nun den Wert 4
```

Zeiger und Referenzen: Beispiel 1

```
int x = 5;
int y = 7;
int& r = x;              // r ist ein Alias für die Variable x

r = y;                   // x erhält den Wert von y

int *pX = &x;            // pX ist die Adresse der Variablen x
int *pY = &y;
int *pR = &r;            // pR ist gleich pX
```

Zeiger und Referenzen: Beispiel 2

```
int x = 2;
int y = 9;

int *p = &x;          // p enthält die Adresse von x
int*& r = p;          // r ist ein Alias für p

*r = 4;               // x erhält den Wert 4
r = &y;               // p erhält die Adresse von y
p = &x;               // p erhält die Adresse von x

cout << *r << endl;   // welcher Wert wird ausgegeben?
```

Klassen

- struct
  - in C: Verbund (Record) von verschiedenen Datenfeldern
  - in C++: öffentliche Klasse (alle Members sind public per Default)

```
struct Point {
    int m_x, m_y;
    void setY(int y) { m_y = y; }
};
```

- class
  - nur in C++: alle Members sind private per Default

```
class Person {
    char m_name[20];
    int m_alter;
public:
    char * getName() { return m_name; }
};
```

Instanzen

- Erzeugen von Instanzen
  - Beispiele

```
Point pnt1;                          // auf Stack
Point *pnt2 = new Point();           // auf Heap
Person pers1;                        // auf Stack
Person *pers2 = new Person();        // auf Heap
Person& refP = pers1;
```

- Zugriff auf Instanzvariablen und Instanzmethoden
  - Beispiele

```
pnt1.m_x = 3;        pnt1.setY(pnt1.m_x);
pnt2->m_x = 4;       pnt2->setY(pnt2->m_x);
char *name = pers1.getName();
name = pers2->getName();
name = refP.getName();
```

Parameterübergabe

- In welcher Art sollen Objekte an Methoden übergeben werden?
  - per value: Daten werden kopiert
  - per reference: Zeiger oder Referenz wird übergeben
    - ist in C++ auch bei einfachen Datentypen möglich
- Grundsatz
  - Datentypen mit weniger oder gleichviel Speicher wie zwei Zeiger werden üblicherweise per value übergeben
- Per Referenz: Zeiger oder Referenzen verwenden?
  - Übergabe per Referenz ist „eleganter“
  - Referenzen benutzen den einfacheren Punkt-Operator anstatt „->“
  - wenn Zeiger schon vorhanden, dann üblicherweise Zeiger verwenden

Neue Zeigerobjekte (C++11)

- std::unique_ptr<T>
  - Zeigerobjekt ist der Besitzer des Objektes, auf welches verwiesen wird
  - pro Objekt existiert höchstens ein einziger unique_ptr
  - das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts zerstört
- std::shared_ptr<T>
  - Zeigerobjekt beinhaltet einen Referenzzähler
  - mehrere Zeigerobjekte können auf das gleiche Objekt zeigen
  - das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts nur dann zerstört, wenn keine weiteren Zeigerobjekte aufs gleiche Objekt zeigen
- std::weak_ptr<T>
  - zum Aufbrechen von zyklischen Abhängigkeiten

Smart Pointers (C++11)

- Prinzip
  - spezielle Zeigerobjekte verwalten Heap-Adressen
  - mittels Referenzzähler wird festgehalten, wie viele Zeigerobjekte auf das gleiche Objekt auf dem Heap zeigen
  - im Destruktor des Zeigerobjektes wird der Referenzzähler überprüft und das Objekt auf dem Heap automatisch gelöscht, wenn keine weiteren Zeigerobjekte mehr auf das gleiche Objekt zeigen
- Ziel
  - der Umgang mit den Zeigerobjekten muss annähernd so einfach sein, wie der Umgang mit gewöhnlichen Zeigern, d.h. der Benutzer soll nichts mit dem Referenzzähler zu tun haben
- Vorteil gegenüber Garbage Collector
  - Speicher wird sofort frei gegeben, sobald er nicht mehr benötigt wird
  - Umgang funktioniert so einfach wie bei lokalen Objekten auf dem Stack

Smart Pointers: Beispiel 1

```
shared_ptr<string> s;
{
    auto p = shared_ptr<string>(new string("shared"));
    // auto p = make_shared<string>(“shared”);
    cout << *p << endl;
    cout << "is p unique? " << boolalpha << p.unique() << endl;
    s = p;
    cout << "is p unique? " << boolalpha << p.unique() << endl;
    cout << "is s unique? " << boolalpha << s.unique() << endl;
    // Speicher wird am Ende dieses Blocks NICHT frei gegeben
}
cout << "is s unique? " << boolalpha << s.unique() << endl;
cout << *s << endl;
```

Smart Pointers: Beispiel 2

```
struct Object;
struct User {
    shared_ptr<Object> obj;
    string name;
    User(string n) : name(n) {}
    ~User() { cout << name << endl; }
};
```

```
struct Object {
    int val;
    weak_ptr<User> owner;
    Object(int v) : val(v) {}
    ~Object() { cout << val << endl; }
};
```

```
{
    auto peter = shared_ptr<User>(new User("Peter"));
    auto vera = make_shared<User>("Vera");
    peter->obj = unique_ptr<Object>(new Object(1));
    peter->obj->owner = peter;
    vera->obj = peter->obj;
}
```