Programmieren in C++

Zeiger und Arrays

Inhalt

- Zeiger in C/C++
- Smart-Pointers in C++
- C-Array und C-String in C/C++
- Arrays und Strings in C++
- Zeichentypen in C++
- Rohstringliterale in C++

Zeiger und Adressoperator

- Zeiger (Pointer)
  - ein Zeiger zeigt auf eine Speicherstelle des (virtuellen) Adressraums
  - Speicherbedarf eines Zeigers: x86: 32 Bit, x64: 64 Bit
  - Zeiger sind stark typisiert
  - von jeder Variable, Funktion, Methode und jedem Objekt kann mit dem Adressoperator & zur Laufzeit die Adresse (Speicherstelle) abgefragt werden; das Resultat einer solchen Abfrage ist ein Zeiger
  - über den Dereferenzierungsoperator * kann vom Zeiger auf die Variable, Funktion, Methode oder das Objekt zugegriffen werden
  - Java: eine Referenz in Java entspricht etwa einem Zeiger in C++
- Einfaches Beispiel

```
int x = 5;          // x wird direkt mit 5 initialisiert
int* px = &x;       // initialisierter Zeiger auf Integer x
*px = 7;            // Wert von x wird indirekt auf 7 gesetzt
```

Eigenschaften von Zeigern

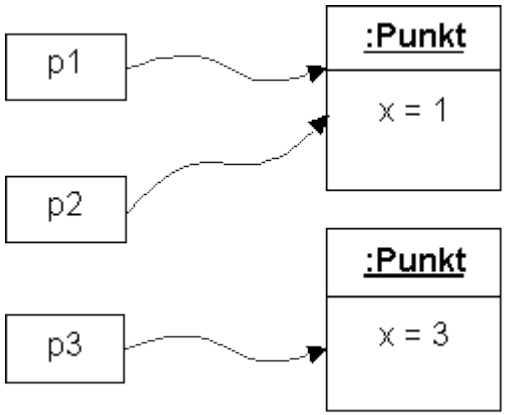
- haben einen Typ „Zeiger auf …“
  - soll eine Zeigervariable auf eine Instanz einer Klasse C zeigen, so muss der Typ der Zeigervariablen zur Klasse C zuweisungskompatibel sein

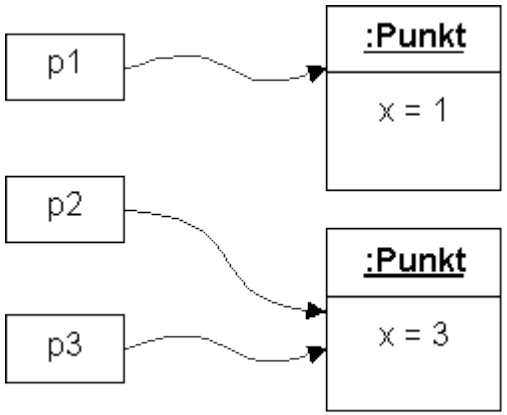
```
Point pnt;
Person pers("Peter", 21);
Point *p1 = &pnt;
Point *p2 = &pers;          // nicht zuweisungskompatibel
```

- zeigen auf gültige Speicheradressen, z.B.
  - dynamisch allozierte Objekte auf dem Heap
  - aufs erste Element von C-Arrays bzw. C-Strings
  - auf statische Variablen und Objekte (Achtung Lebensdauer!)
- zeigen auf ungültige Speicheradressen
  - nullptr
  - nicht initialisierten Speicherbereich

Zuweisungen bei Zeigervariablen

```
Point *p1 = new Point();  // p1 zeigt auf neu erstelltes Objekt auf dem Heap
Point *p2;                // p2 ist ein nichtinitialisierter Zeiger
Point *p3 = new Point();  // p3 zeigt auf neues Point-Objekt auf dem Heap
p2 = p1;                  // p2 zeigt zum gleichen Objekt wie p1
p2 = p3;                  // Adresse p3 wird nach p2 kopiert
// Achtung: die Punktdaten werden hier nicht kopiert
```





Instanzen und Instanzmethoden

- Erzeugen von Klasseninstanzen

  Beispiele

```
Point pnt1;                          // Punktobjekt auf dem Stack
Person pers1("Anna", 22);            // Person auf dem Stack
Point *pnt2 = new Point();           // Punktobjekt auf dem Heap
Person *pers2 = new Person("Urs", 21);   // Person auf dem Heap
```

- Zugriff auf Instanzvariablen bzw. Instanzmethoden

  Beispiele

```
pnt1.m_x = 3;    // direkter Zugriff auf Instanzvariable m_x
(*pnt2).m_x = 4; // indirekter Zugriff auf Instanzvariable m_x
pnt2->m_x = 5;   // vereinf. Schreibweise des indirekten Zugriffs
auto d1 = pnt1.dist(*pnt2);
auto d2 = pnt2->dist(pnt1);
```

Zeiger und const

- Schlüsselwort const in Verbindung mit Zeigern
  - was soll ausgedrückt werden?
    - Adresse ist unveränderbar (Wert des Zeigers 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;        // oder: int const *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;  // oder: int const * const p = &x;
```

Zeiger und Referenzen

```
int x;
int& rx = x;     // rx ist ein Alias für die Variable x
int* px = &x;    // px ist ein Zeiger auf x
px = &rx;        // px ist auch ein Zeiger auf x

int k;
int *pk = &k;    // das Ampersand (&) ist der Adressoperator
int*& rpk = pk; // rpk ist ein Alias für den Zeiger pk
*rpk = 4;        // die Variable k kriegt den Wert 4

int a = 2, b = 9;
int *pa = &a, *& rpa = pa;
*rpa = 4; rpa = &b; pa = &a;
cout << *rpa << endl;       // welcher Wert wird ausgegeben?
```

Smart Pointers (C++)

- Prinzip
  - spezielle Zeigerobjekte verwalten 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 soll so einfach sein, wie der Umgang mit Rohzeigern, d.h. der Benutzer soll nichts mit dem Referenzzähler zu tun haben
  - Verzicht auf explizite Speicherallokation (`new`) und -freigabe (`delete`)
- Vorteil gegenüber Garbage Collector (Performanz)
  - Speicher wird sofort frei gegeben, sobald er nicht mehr benötigt wird
  - keine aufwendige Suche von nicht mehr benötigten Objekten
  - Umgang funktioniert so einfach wie bei lokalen Objekten auf dem Stack

Ownership-Konzept

- Heap-Objekt hat genau einen Besitzer
  - `std::unique_ptr<T>`
    - pro Objekt existiert höchstens ein einziger Besitzer
    - unique_ptr ist der Besitzer des Objektes, auf welches verwiesen wird
    - wird als Returntyp von Factories verwendet
    - das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts zerstört
- Heap-Objekt kann mehrere Besitzer haben
  - `std::shared_ptr<T>`
    - mehrere Zeigerobjekte können auf das gleiche Objekt zeigen
    - shared_ptr benutzt Referenzzähler
    - das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts nur dann zerstört, wenn keine weiteren shared_ptr aufs gleiche Objekt zeigen
  - `std::weak_ptr<T>`
    - wie shared_ptr, aber der weak_ptr zählt nicht als Besitzer des Heap-Objekts
    - im Referenzzähler wird zwischen shared_ptr und weak_ptr unterschieden
    - wird zum manuellen Aufbrechen von zyklischen Abhängigkeiten benötigt

Unique-Pointer: Beispiel

```
unique_ptr<Object> factory(int val) {
   return make_unique<Object>(val);
}
void unique_test() {
   auto up = factory(1);
   shared_ptr<Object> sp = factory(2);
   assert(sp.use_count() == 1);
   auto up2 = up;                  // nicht erlaubt, weil unique_ptr
                                   // keinen Zuweisungsoperator hat
   shared_ptr<Object> sp2 = up;    // nicht erlaubt, weil up der
                                   // alleinige Besitzer sein muss
   sp = move(up);                  // up überträgt die Ownership an sp
   assert(sp.use_count() == 1);
   assert(up.get() == nullptr);
}
```

Shared-Pointer: Beispiel

```
#include <memory>

{
   shared_ptr<string> s; // s.rc = 0
   {
      auto sp = make_shared<string>("shared"); // sp.rc = 1
      cout << *sp << endl;
      cout << "is sp unique? " << sp.use_count() << endl;
      s = sp; // s.rc = sp.rc = 2
      cout << "is sp unique? " << sp.use_count() << endl;
      cout << "is s unique? " << s.use_count() << endl;
      // Speicher des Strings wird am Ende dieses Blocks NICHT frei
      // gegeben, weil s noch auf das String-Objekt zeigt
   } // s.rc = 1
   cout << "is s unique? " << s.use_count() << endl;
   cout << *s << endl;
}
```

Shared-Pointers: Beispiel

```
struct Object;
struct User {
    shared_ptr<Object> m_obj;
    string m_name;
    User(string n) : m_name(n) {}
    ~User(){ cout << m_name << endl; }
};
{
    auto peter = make_shared<User>("Peter");      // peter.rc = 1
    auto vera = make_shared<User>("Vera");        // vera.rc = 1
    auto object = make_shared<Object>(1);         // object.rc = 1
    peter->m_obj = object;                        // object.rc = 2
    object->m_owner = peter;                      // peter.rc = 2
    vera->m_obj = object;                         // object.rc = 3
}   // object.rc = 2   vera.rc = 0   object.rc = 1   peter.rc = 1
```

```
struct Object {
    int m_val;
    shared_ptr<User> m_owner;
    Object(int v) : m_val(v) {}
    ~Object() { cout << m_val << endl; }
};
```

Shared- und Weak-Pointers: Beispiel

```
struct Object;
struct User {
    weak_ptr<Object> m_obj;
    string m_name;
    User(string n) : m_name(n) {}
    ~User(){ cout << m_name << endl; }
};
{
    auto peter = make_shared<User>("Peter");      // peter.rc = 1,0
    auto vera = make_shared<User>("Vera");        // vera.rc = 1,0
    auto object = make_shared<Object>(1);         // object.rc = 1,0
    peter->m_obj = object;                        // object.rc = 1,1
    object->m_owner = peter;                      // peter.rc = 1,1
    vera->m_obj = object;                         // object.rc = 1,2
}   // object.rc = 0,2   peter.rc = 1,0   vera.rc = 0,0   peter.rc = 0,0
```

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

Eindimensionale C-Arrays

- Grundsätze
  - Länge des Arrays wird nicht im Array abgespeichert
  - die Länge ist dem Compiler nur im Sichtbarkeitsbereich der Definition des Arrays bekannt
  - sehr grosse Arrays sollen auf dem Heap (dynamisch) angelegt werden
- statische Erzeugung
  - Wenn die Arraylänge zur Kompilationszeit bekannt und konstant ist, dann kann das Array auf dem Stack angelegt werden
  - Beispiel

```
char text[100];
```

- dynamische Erzeugung
  - Array wird zur Laufzeit auf dem Heap angelegt
  - Beispiel

```
int len = …;                          // len kann, muss aber nicht konstant sein
char * const text = new char[len]; // new liefert einen konstanten Zeiger zurück
delete[] text;                        // Speicherplatz des Arrays wird freigegeben
```

C-Strings

- Was ist ein C-String?
  - ein eindimensionales Character-Array mit 0-Terminierung
  - Ende der gültigen Zeichenkette ist durch ein '\0'-Character gekennzeichnet
  - die 0-Terminierung benötigt ein zusätzliches Byte
- Unterschied zu anderen Arrays
  - vereinfachte Initialisierung erlaubt
    - char s[] = "Das ist ein Test."; // String-Schreibweise anstatt Initialisierungsliste
    - s zeigt auf eine Kopie des String-Literals "Das ist ein Test."
    - sizeof(s) gibt den Speicherbedarf des Strings nur im Sichtbereich der Definition zurück, ausserhalb wird der Speicherbedarf des Zeigers s zurückgegeben
  - implizite Konstante
    - const char *t = "Das ist ein Test.";
    - t zeigt direkt auf den konstanten String der Länge 17 + 1 Byte für 0-Terminierung
    - sizeof(t) gibt die Anzahl Bytes des Zeigers t zurück

C-Array/Strings als Funktionsparameter

- Annahme: Funktion liegt ausserhalb des Sichtbarkeitsbereichs der Definition des Arrays
  - Länge des Arrays ist nicht bekannt und sollte als Parameter mitgegeben werden
  - sizeof kann nicht die Länge ermitteln, sondern nur die Anzahl Bytes des Zeigers
  - 2 gleichwertige Schreibweisen
    - void print(char *s) { … }
    - void print(char s[]) { … } // ist zu bevorzugen, weil Array besser ersichtlich ist

- keine Spezialbehandlung für C-Strings

```
void foo(char s[]) {
    s[0] = 'a';
}
```

```
char s1[] = "ABC";
const char *s2 = "DEF"; // ohne const deprecated

foo(s1);
foo(s2);        // Compiler-Error, weil s2 const ist
```

Zeigerarithmetik

- Voraussetzung
  - Zeigertyp: in einer Zeigervariablen (Zeiger) wird eine Speicheradresse verwaltet
- Idee
  - aus bestehender Speicheradresse wird eine neue Adresse berechnet
- Erlaubte Operationen
  - +, +=, ++
  - -, -=, --
  - Ergebnis ist vom gleichen Zeigertyp
  - +1 bedeutet nicht + 1 Byte, sondern + Anzahl Bytes des Zielobjekts des Zeigers
- Typischer Einsatz
  - durch die Bildpunkte eines Rasterbildes (Arrays) iterieren

Zeigerarithmetik: Rasterbild

```
constexpr int width = 41;
constexpr int height = 31;
const int bytesPerLine = ((width + 3)/4)*4;     // verbessertes Alignement

uint8_t *const grayImage = new uint8_t[bytesPerLine*height];
uint8_t *row = grayImage;

for(int v=0; v < height; v++) {
    for(int u=0; u < width; u++) {
        row[u] = (uint8_t)(u + v);          // *(row + u) = (uint8_t)(u + v);
    }
    row += bytesPerLine;                    // Zeigerarithmetik
}

delete[] grayImage;
```

C++ Arrays (statisch)

- class array<T,S> mit fester Grösse S
  - generische Klasse aus der STL
  - kapselt ein C-Array fixer Länge und bietet ein paar nützliche Array-Methoden
  - keine Unterscheidung zwischen Array-Länge und Kapazität
- Beispiel

```
#include <array>
#include <string>

constexpr size_t size = 4;
array<string, size> names = { "adam", "berta", "carlo", "doris" };
array names2 { "adam", "berta", "carlo", "doris" };
int i = 0;
for (const auto& s : names) {
   cout << i++ << ": " << s << endl;
}
```

C++ Vektoren (halbdynamisch)

- class vector<T>
  - generische Klasse aus der STL, entspricht der ArrayList aus Java
  - Unterscheidung zwischen Länge und Kapazität

- Beispiel

```
vector<string> vnames = { "adam", "berta", "carlo", "doris" };
vector<shared_ptr<string>> vsp;
vsp.reserve(vnames.size());// allocates enough memory on heap

for (const auto& s : vnames) {
   vsp.push_back(make_shared<string>(s + ':' +
      to_string(s.length())));
}
for (size_t i = 0; i < vnames.size(); i++) {
   cout << vnames[i] << endl;
   cout << *vsp[i] << endl;
}
```

C++ Strings

- class string ist gleich basic_string<char>
  - #include <string>
- Beispiel

```
auto name = "Andrea"s;          // ist ein C++-String und kein C-String
                                // wegen dem suffix s

name.size();                    // Anzahl Zeichen
name.length();                  // Anzahl Zeichen
name[2];                        // direkter Zeichenzugriff
name.c_str();                   // Konverter zu nullterminiertem C-String
name.begin();                   // Iteratoren
name.substr(2, 2);              // Substring(pos, len)
name.find("re");                // Suchalgorithmus
```

C++ string_view

- Wrapper für ein String-Literal und seine Länge
  - besteht üblicherweise nur aus zwei Attributen
    - const char* data; // Zeiger zu einer konstanten Zeichenkette
    - size_t size; // Anzahl Zeichen
- Eigenschaften
  - das string_view Objekt ist nicht der Besitzer der Zeichenkette
    - die Zeichenkette wird nicht von string_view angelegt
    - der Speicher der Zeichenkette wird nicht freigegeben
  - die Konstruktoren und Methoden benötigen keine Ausführungszeit
    - die Objekte und Rückgabewert sind constexpr
- Einsatz
  - kann als effizienter Ersatz eines C-Strings verwendet werden
  - die Länge des C-Strings muss nicht separat an eine Methode übergeben werden

C++ span<T>

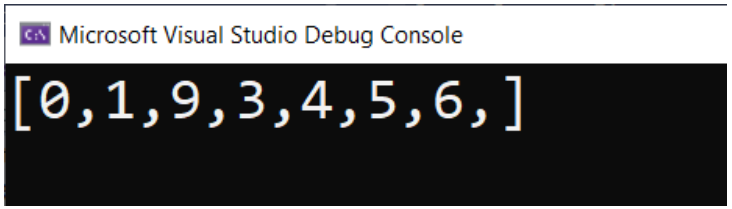
- Kapselung einer Sequenz von Daten und deren Länge
  - typische Datensequenzen sind:
    - C-Array
    - std::array
    - std::vector
  - ähnliches Prinzip wie bei string_view
    - Daten der Sequenz dürfen aber verändert werden
  - typischer Einsatz
    - zur Datenübergabe an Funktionen
- Beispiele
  - span<int> s1; // Sequenz von int's
  - span<const int> s2; // Sequenz von nicht veränderbaren int's
  - span<int, 5> s3; // Sequenz von exakt 5 int's

C++ span<T>: Beispiel

```
std::vector<int> v{ 0,1,2,3,4,5,6 };
std::span<int> s = v;

auto sub = s.subspan(2, 4);
sub[0] = 9;

std::cout << '[';
for (auto& i : v) {
  std::cout << i << ',';
}
std::cout << ']' << std::endl;
```



C++ Zeichentypen

- Standardzeichentypen (in Standardbibliothek voll unterstützt)
  - const char *s = "abcd"; 1 Byte pro char
  - const wchar_t *s = L"αβγδ"; mehrere Bytes pro Character (z.B. UTF-16)
- String-Repräsentationen
  - const char8_t *s = u8"αβγδ"; UTF-8 String-Repräsentation
  - const char16_t *s = u"αβγδ "; UTF-16 String-Repräsentation
  - const char32_t *s = U"αβγδ "; UTF-32 String-Repräsentation
- Unicode-Codepoints
  - 16 Bit Unicode-Codepoints: \u1234 (4-stelliger Hex-Code)
  - 32 Bit Unicode-Codepoints: \U00123456 (8-stelliger Hex-Code)

C++ String-Typen

```
// UTF-8 (neuer MSVC-Standard)
string s = "ab\u1234\U00103456äö👦👧";

// mehrere Bytes pro Character
wstring s = L"ab\u1234\U00103456äö👦👧";

// UTF-8 String-Repräsentation
u8string s = u8"ab\u1234\U00103456äö👦👧";

// UTF-16 String-Repräsentation
u16string s = u"ab\u1234\U00103456äö👦👧";

// UTF-32 String-Repräsentation
u32string s = U"ab\u1234\U00103456äö👦👧";
```

C++ Rohstringliterale

- Ausgangslage
  - Benutzung von bestimmten Zeichen in Strings ist erschwert
  - Beispiel: Anführungszeichen, Backslashes und Sonderzeichen müssen speziell markiert werden: "abc \"def\" \\ghi\\ \n"
- Rohstringliterale
  - Anführungszeichen, Backslashes und Zeilenumbrüche können ohne weitere Vorkehrungen vorkommen
  - Rohstringliterale beginnen mit R"(und enden mit)"
  - sollte)" innerhalb des Strings vorkommen, so muss eine eigene Markierung verwendet werden, z.B. R"*(abc "(def)" ghi)*"
- Beispiel

```
const char *s = R"—(<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type">
)—";
```