Vorkurs C++ Programmierung

Zeiger

Referenzen

UNIVERSITÄT SIEGEN

Letzte Stunde

- Arrays als Zusammenfassung von Daten gleichen Typs
- Funktionen als Zusammenfassung von Anweisungen, die man wiederholt braucht
- Parameterübergabe:
  - Call-by-Value-Problematik
    - Kopieraufwand
    - Änderungen bleiben nur lokal!

Letzte Stunde: Call by Value...

Rückblick...

```
#include <iostream>

void quadrat (int i)
{
   i = i * i;
}

int main (int argc, char* argv[])
{
   int s = 2;
   quadrat(s);
   std::cout << s << std::endl;
   return 0;
}
```

Ausgabe: 2

Call by Value:

- „intern“ wird der Variablenwert beim Aufruf in eine typgleiche Variable kopiert
- bei umfangreichen Datentypen entsteht großer Kopieraufwand
- eine Veränderung innerhalb der Funktion hat keine Auswirkung nach außen, d.h. der Wert der Variablen *im Aufruf* bleibt unverändert!

Call by Value

- Änderungen sind nur lokal
- Kopieraufwand
  - wäre bei kleinen Datenmengen zu vernachlässigen, aber...
- Vorgriff:
  - *Klassen* (eigene, selbst definierte Typen) brauchen *viel mehr Speicherplatz*, nicht vergleichbar mit den Standarddatentypen.
  - Übergabe dieser Typen als Parameter an Funktionen oder Rückgabe über Returnwerte erfordert *Kopieren*

→ Werden diese großen Typen kopiert, fordert das Programm zu viel Speicher, es wird langsam und ist ineffizient!

→ Kopieren sollte vermieden werden

Call by Reference

Was ist die Alternative?

Call by Reference: Parameter ist ein Referenz- oder Zeigertyp!

- Eine Veränderung des referenzierten Wertes innerhalb der Funktion hat eine Auswirkung nach außen
- Der Wert der referenzierten Variablen wird geändert!

Aber was ist eine Referenz, was ist ein Zeiger?

Referenzen (references)

- Eine Referenz ist die Adresse, an der eine Variable im Speicher liegt
- Mit dem Adressoperator **&** kann man diese Adresse erfragen.
- Deklaration: ***type &refName = varName;***
  - Die Variable **varName** muss schon initialisiert sein
  - **refName** verweist auf die Adresse der Variable **varName**
  - ***type*** ist der Typ der Variable **varName**
- Beispiel:

```
int i = 10;        //Deklaration einer Variablen
int &r = i;        //Deklaration einer Referenz auf die
                   //Adresse von i
r = 5;             //Änderung von r
std::cout << i << std::endl;
```

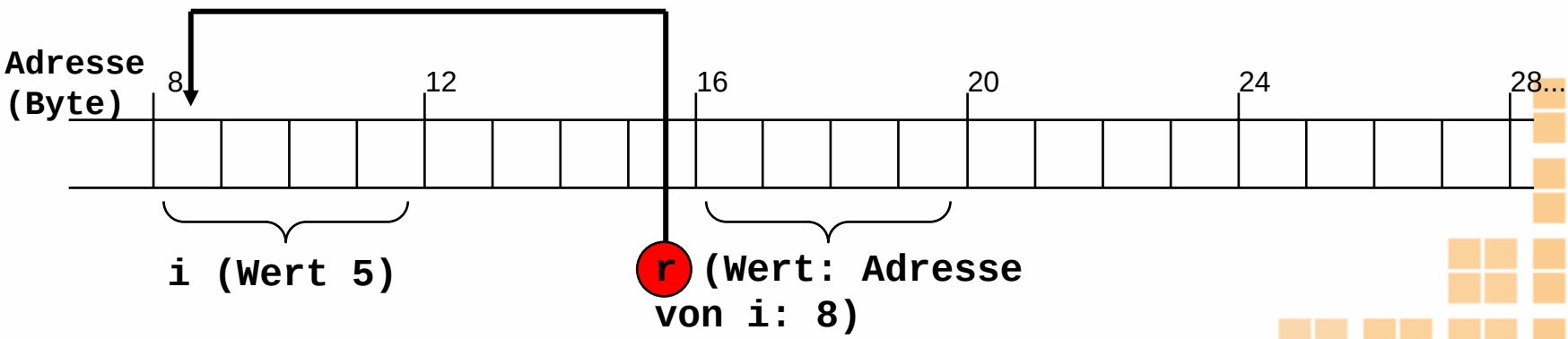
Ausgabe: i = 5

Referenzen (references)

- Eine Referenz ist die Adresse, an der eine Variable im Speicher liegt

Beispiel:

```
int i = 10;        //Deklaration einer Variablen
int &r = i;        //Deklaration einer Referenz auf die
                   //Adresse von i
r = 5;             //Änderung von r
std::cout << i << std::endl;
```



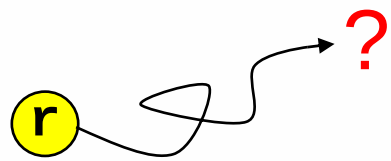
Referenzen (references)

```
// Deklaration
int k = 10;
int i = 5;
int copy_k = k;   // setze copy_k auf den Wert von k:
                  // copy_k = 10
int& r_i = i;     // erstelle Referenz auf i

// Zuweisung
copy_k = 20;      // setze copy_k = 20 (k bleibt 10)
r_i = 20;         // r_i referenziert noch immer i.
                  // i wird auf 20 gesetzt
r_i = k;          // Wertzuweisung mit k (=10); i wird 10
```

Beachte: Es ist nicht möglich eine Referenz zu deklarieren, ohne direkt eine referenzierte Variable anzugeben, weil eine Referenz auf eine Speicherstelle zeigen **muss:**

```
int& r_i; //nicht erlaubt!
```



Zeiger (pointer)

- Eine *Referenz* beinhaltet die Adresse der ihr zugewiesenen Variablen. Eine Referenz ist nicht veränderlich.
- Auch **Zeiger** sind Adressen, können aber im Gegensatz zu Referenzen „während ihrer Existenz“ auf unterschiedliche Speicherstellen zeigen

Deklaration:

type *ptrName; **bzw.** ***type *ptrName = &varName;***

- Deklaration einer Zeigervariablen ***ptrName***, ggf. mit Initialisierung
- ***ptrName*** verweist auf Variable vom Typ ***type***
- Der ***** macht die Variable zur Adressvariablen.

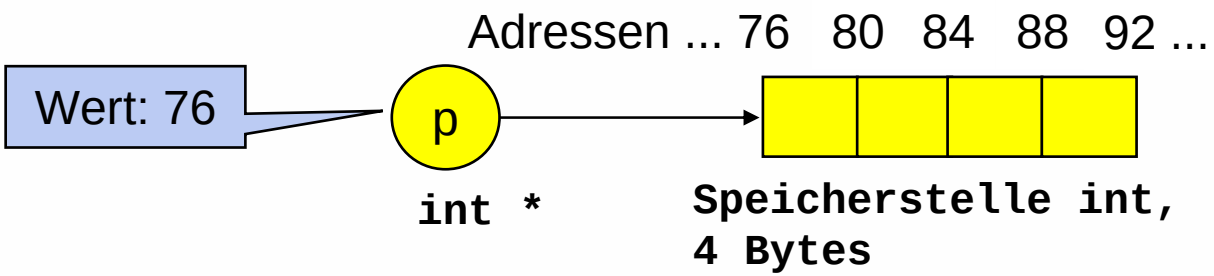
Beispiel:

```
int i = 5;        //normale int-Variable
int *p_i = &i;    //Zeiger p_i zeigt jetzt auf die
                  //Speicherstelle von i
```

Zeiger (pointer)

Beispiel:

```
int i = 5;        //normale int-Variable
int *p = &i;      //Zeiger p zeigt jetzt auf die
                  //Speicherstelle von i
```



Der Typ des Zeigers bezeichnet den Inhalt des Speichers, auf den der Zeiger verweisen kann

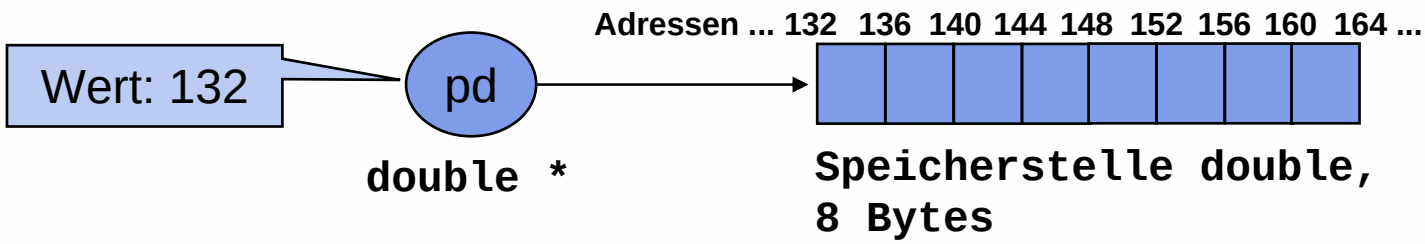
→ ein **`int`**-Pointer kann nur auf eine Speicherstelle zeigen, an der eine **`int`**-Variable gespeichert ist! Dies gilt für andere Datentypen entsprechend.

→ ein falscher Zeigertyp würde zur falschen „Interpretation“ des Speicherinhalts und somit zu Fehlern führen

Zeiger (pointer)

Beispiel:

```
double d = 1.3; //normale double-Variable
double *pd = &d;//Zeiger pd zeigt jetzt auf die
                //Speicherstelle von d
```



Der Typ des Zeigers bezeichnet den Inhalt des Speichers, auf den der Zeiger verweisen kann

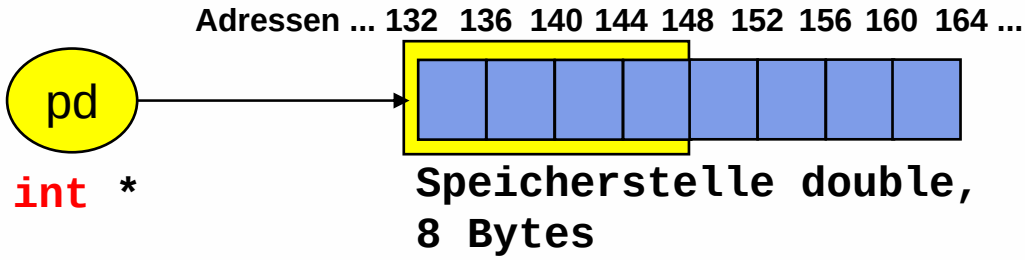
→ ein **`int`**-Pointer kann nur auf eine Speicherstelle zeigen, an der eine **`int`**-Variable gespeichert ist! Dies gilt für andere Datentypen entsprechend.

→ ein falscher Zeigertyp würde zur falschen „Interpretation“ des Speicherinhalts und somit zu Fehlern führen

Zeiger (pointer): *Falsche Verwendung!*

Beispiel:

```
double d = 1.3; //normale double-Variable
int *pd = &d;   //Fehler!
                //pd ist ein int-pointer
```



Der Typ des Zeigers bezeichnet den Inhalt des Speichers, auf den der Zeiger verweisen kann

→ ein **`int`**-Pointer kann nur auf eine Speicherstelle zeigen, an der eine **`int`**-Variable gespeichert ist! Dies gilt für andere Datentypen entsprechend.

→ ein falscher Zeigertyp würde zur falschen „Interpretation“ des Speicherinhalts und somit zu Fehlern führen

Zeiger (pointer): Verwendung

- Wenn man über einen Zeiger auf den Inhalt des Speichers zugreifen möchte, braucht man dazu wieder einen Operator, den Dereferenzierungsoperator *****

```
int main (int argc, char* argv[])
{
   int i;
   int *p;
   p = &i;
   i = 7;
   std::cout << *p << std::endl;
   return 0;
}
```

Ausgabe: 7 → Das ist der Wert, der an dieser Stelle steht.

Zeiger (pointer): Verwendung

- Wenn man über einen Zeiger auf den Inhalt des Speichers zugreifen möchte, braucht man dazu wieder einen Operator, den Dereferenzierungsoperator *****

```
int main (int argc, char* argv[])
{
   int i;
   int *p;
   p = &i;
   i = 7;
   std::cout << p << std::endl;
   return 0;
}
```

Wenn ich den Dereferenzierungsoperator weglasse...

Ausgabe: **0x22cce4** → Das ist eine Speicheradresse!

Zeiger (pointer): Verwendung

- Der Dereferenzierungsoperator kann auch zum Beschreiben der Variablen (der Speicherstelle) verwendet werden:

```
int main (int argc, char* argv[])
{
   int i;
   int *p;
   p = &i;
   i = 7;
   std::cout << *p << std::endl;       //Ausgabe: 7
   *p = 1234;                          //Überschreiben von *p
   std::cout << *p << std::endl;       //Ausgabe 1234
   std::cout <<  i << std::endl;       //Ausgabe 1234
   return 0;
}
```

Wozu Referenzen und Pointer?

- ...zurück zum Anfang der Stunde...

Call by Value...

Rückblick...

```
#include <iostream>

void quadrat (int i)
{
   i = i * i;
}

int main (int argc, char* argv[])
{
   int s = 2;
   quadrat(s);
   std::cout << s << std::endl;
   return 0;
}
```

Ausgabe: 2

Call by Value:

- „intern“ wird der Variablenwert beim Aufruf in eine typgleiche Variable kopiert
- eine Veränderung innerhalb der Funktion hat keine Auswirkung nach außen, d.h. der Wert der Variablen *im Aufruf* bleibt unverändert!
- bei umfangreichen Datentypen entsteht großer Kopieraufwand

Jetzt: Call by Reference

Rückblick...

```
#include <iostream>

void quadrat (int &i)
{
   i = i * i;
}

int main (int argc, char* argv[])
{
   int s = 2;
   quadrat(s);
   std::cout << s << std::endl;
   return 0;
}
```

Funktion erwartet jetzt eine Referenz!

Ausgabe: 4

Call by Reference:

- kein Kopieren mehr, sondern Übergabe der Speicheradresse
- kein Kopieraufwand mehr
- eine Veränderung innerhalb der Funktion hat Auswirkung nach außen:
- Der Wert der Variablen im Aufruf wird geändert
- Handhabung wie Variablen
- Arrays werden automatisch per reference übergeben.

Wozu Referenzen und Pointer?

Grundsätzlich:

- Referenzen und Zeiger haben eine feste Größe von 32 (bzw. 64) Bit
- Bei Typen, die viel Speicher belegen, ist es effizienter, als Funktionsparameter eine Referenz zu übergeben als die Variable zu kopieren

Referenzen (&) sind in ihrer Handhabung wie Variablen

- (Deshalb keine Änderung des Aufrufs im Beispiel)
- Keine Dereferenzierung mittels * notwendig
- referenzieren immer eine und dieselbe Variable

  → keine Speicherzugriffsfehler

Wozu Referenzen und Pointer?

Grundsätzlich:

- Referenzen und Zeiger haben eine feste Größe von 32 (bzw. 64) Bit
- Bei Typen, die viel Speicher belegen, ist es effizienter, als Funktionsparameter eine Referenz zu übergeben als die Variable zu kopieren

Zeiger (*):

- Nutzung der Zeiger-Arithmetik möglich (kommt noch...)
- Können auf verschiedene Variablen verweisen

  → Speicherzugriffsfehler sind möglich

Wann Referenzen, wann Pointer?

Zusammenfassung

- Call by Value vs. Call by Reference
- Referenzen
  - zeigen auf eine Speicherstelle (und nur auf diese)
  - müssen immer eine initialisierte Variable referenzieren
  - keine Dereferenzierung, Verwendung wie Variablen („Aliasname“)
- Pointer
  - zeigen auf eine Speicherstelle (sind veränderlich)
  - Dereferenzierung mit *, dadurch kann der Inhalt an der Speicherstelle gelesen und geschrieben/geändert werden

→Wenn möglich, Referenzen oder Zeiger als Parameter an Funktionen übergeben

→Call-by-Reference.

Nächste Stunde...

- Speicherverwaltung