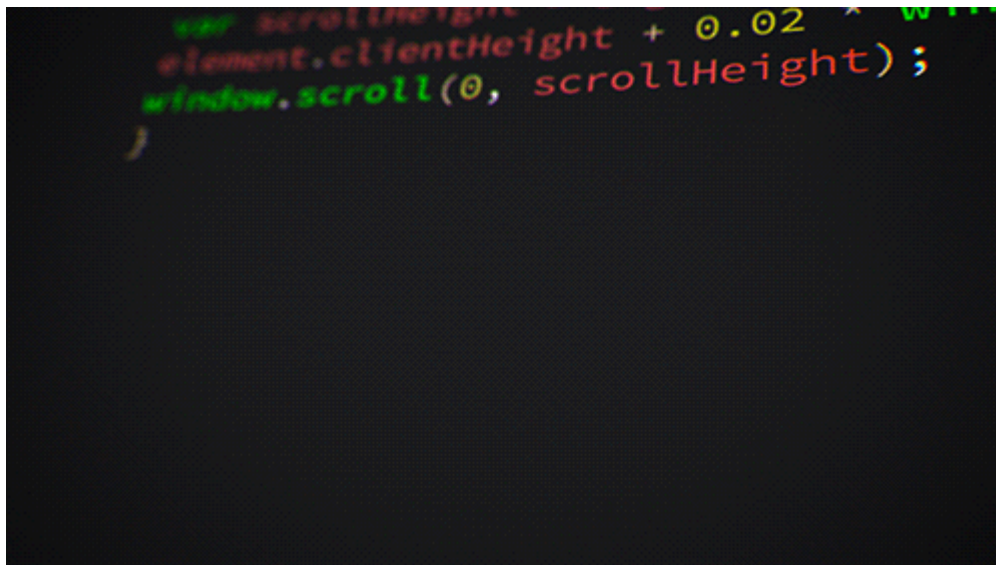


# Grundlagen der Sprache C++

Parameter	Kursinformationen
Veranstaltung:	<u>Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten</u>
Semester	Wintersemester 2025/26
Hochschule:	Technische Universität Freiberg
Inhalte:	Array, Zeiger und Referenzen
Link auf Repository:	<a href="https://github.com/TUBAF-lfl-LiaScript/VL_EAVD/blob/master/03_ArrayZeigerReferenzen.md">https://github.com/TUBAF-lfl-LiaScript/VL_EAVD/blob/master/03_ArrayZeigerReferenzen.md</a>
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf



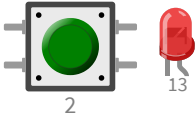
---

## Fragen an die heutige Veranstaltung ...

- Was ist ein Array?
- Wie können zwei Arrays verglichen werden?
- Erklären Sie die Idee des Zeigers in der Programmiersprache C/C++.
- Welche Gefahr besteht bei der Initialisierung von Zeigern?
- Was ist ein `NULL`-Zeiger und wozu wird er verwendet?

## Wie weit waren wir gekommen?

Aufgabe: Die LED blinkt im Beispiel 10 mal. Integrieren Sie eine Abbruchbedingung für diese Schleife, wenn der grüne Button gedrückt wird. Welches Problem sehen Sie?



### ButtonLogic.cpp

```
1 void setup() {
2   pinMode(2, INPUT);      // Button grün
3   pinMode(13, OUTPUT);
4
5   for (int i = 0; i<10; i++) {
6     digitalWrite(13, HIGH);
7     delay(500);
8     digitalWrite(13, LOW);
9     delay(500);
10  }
11 }
12
13 void loop() {
14 }
```

Sketch uses 996 bytes (3%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

Sketch uses 996 bytes (3%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

## Arrays

Bisher umfassten unsere Variablen einzelne Skalare. Arrays erweitern das Spektrum um Folgen von Werten, die in n-Dimensionen aufgestellt werden können. Ein Array ist eine geordnete Folge von Werten des gleichen Datentyps. Die Deklaration erfolgt in folgender Anweisung:

```
Datentyp Variablenname[Anzahl_der_Elemente];
```

```
int a[6];
```

a[0]      a[1]      a[2]      a[3]      a[4]      a[5]

```
Datentyp Variablenname[Anzahl_der_Elemente_Dim0][Anzahl_der_Elemente_Dim1];
```

```
int a[3][5];
```

	Spalten				
Zeilen	<div>a[0] [0]</div>	<div>a[0] [1]</div>	<div>a[0] [2]</div>	<div>a[0] [3]</div>	<div>a[0] [4]</div>
	<div>a[1] [0]</div>	<div>a[1] [1]</div>	<div>a[1] [2]</div>	<div>a[1] [3]</div>	<div>a[1] [4]</div>
	<div>a[2] [0]</div>	<div>a[2] [1]</div>	<div>a[2] [2]</div>	<div>a[2] [3]</div>	<div>a[2] [4]</div>

**Achtung 1:** Im hier beschriebenen Format muss zum Zeitpunkt der Übersetzung die Größe des Arrays (Anzahl\_der\_Elemente) bekannt sein.

**Achtung 2:** Der Variablenname steht nunmehr nicht für einen Wert sondern für die Speicheradresse (Pointer) des ersten Elementes!

## Deklaration, Definition, Initialisierung, Zugriff

Initialisierung und genereller Zugriff auf die einzelnen Elemente des Arrays sind über einen Index möglich.

## ArrayExample.cpp



```
1 #include <iostream>
2
3 int main(void) {
4     int a[3];          // Array aus 3 int Werten
5     a[0] = -2;
6     a[1] = 5;
7     a[2] = 99;
8     for (int i=0; i<3; i++)
9         std::cout << a[i] << " ";
10    std::cout << "\nNur zur Info " << sizeof(a) << "\n";
11    std::cout << "Zahl der Elemente " << sizeof(a) / sizeof(int) << "\n";
12    return 0;
13 }
```

```
-2 5 99
Nur zur Info 12
Zahl der Elemente 3
```

Schauen wir uns das Ganze noch in einer Animation an: [PythonTutor](#)

Wie können Arrays noch initialisiert werden:

- vollständig (alle Elemente werden mit einem spezifischen Wert belegt)
- anteilig (einzelne Elemente werden mit spezifischen Werten gefüllt, der rest mit 0)

## ArrayExample.cpp



```
1 #include <iostream>
2 #include <iomanip>
3
4 int main(void) {
5     int a[] = {5, 2, 2, 5, 6};
6     float b[5] = {1.01};
7     for (int i=0; i<5; i++) {
8         std::cout << i << " " << a[i] << " / " << std::fixed << b[i] << " ";
9     }
10    return 0;
11 }
```

```
0 5 / 1.010000
1 2 / 0.000000
2 2 / 0.000000
3 5 / 0.000000
4 6 / 0.000000
0 5 / 1.010000
1 2 / 0.000000
2 2 / 0.000000
3 5 / 0.000000
4 6 / 0.000000
```

Und wie bestimme ich den erforderlichen Speicherbedarf bzw. die Größe des Arrays?

#### ArrayExample.cpp

```
1 #include <iostream>
2
3 int main(void) {
4     int a[3];
5     std::cout << "Speicherplatz [Byte] " << sizeof(a) << "\n";
6     std::cout << "Zahl der Elemente " << sizeof(a)/sizeof(int) << "\n";
7     return 0;
8 }
```

```
Speicherplatz [Byte] 12
Zahl der Elemente 3
Speicherplatz [Byte] 12
Zahl der Elemente 3
```

## Fehlerquelle Nummer 1 - out of range

#### ArrayExample.cpp

```
1 #include <iostream>
2
3 int main(void) {
4     int a[] = {-2, 5, 99};
5     for (int i=0; i<=3; i++)
6         std::cout << a[i] << " ";
7     std::cout << "\n";
8     return 0;
9 }
```

```
-2 5 99 -119593728
-2 5 99 1946766080
```

## Anwendung eines eindimensionalen Arrays

Schreiben Sie ein Programm, das zwei Vektoren miteinander vergleicht. Warum ist die intuitive Lösung `a == b` nicht korrekt, wenn `a` und `b` arrays sind?

### ArrayExample.cpp

```
1  #include <iostream>
2
3  int main(void) {
4      int a[] = {0, 1, 2, 4, 3, 5, 6, 7, 8, 9};
5      int b[10];
6      for (int i=0; i<10; i++) // "Befüllen" des Arrays b
7          b[i] = i;
8      for (int i=0; i<10; i++)
9          if (a[i] != b[i])
10             std::cout << "An Stelle " << i << " unterscheiden sich die Arrays!";
11     return 0;
12 }
```

```
An Stelle 3 unterscheiden sich die Arrays!
An Stelle 4 unterscheiden sich die Arrays!
An Stelle 3 unterscheiden sich die Arrays!
An Stelle 4 unterscheiden sich die Arrays!
```

Welche Verbesserungsmöglichkeiten sehen Sie bei dem Programm?

## Mehrdimensionale Arrays

Deklaration:

```
int Matrix[4][5];    /* Zweidimensional - 4 Zeilen x 5 Spalten */
```

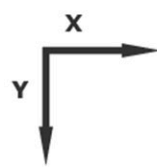
Deklaration mit einer sofortigen Initialisierung aller bzw. einiger Elemente:

```
int Matrix[4][5] = {{1, 2, 3, 4, 5},
                    {6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15},
                    {16, 17, 18, 19, 20}};
```

```
int Matrix[4][4] = {{1,},
                    {1, 1},
                    {1, 1, 1},
                    {1, 1, 1, 1}};

int Matrix[4][4] = {1,2,3,4,5,6,7,8};
```

Initialisierung eines n-dimensionalen Arrays:



	0	1	2	3	4	5	6	7
0								
1								
2		1						
3					3			
4			2					
5								
6							4	
7								

Darstellung der Matrixinhalte für das nachfolgende Codebeispiel <sup>[1]</sup>

#### nDimArray.cpp

```
1  #include <iostream>
2
3  int main(void) {
4      // Initialisierung
5      int brett[8][8] = {0};
6      // Zuweisung
7      brett[2][1] = 1;
8      brett[4][2] = 2;
9      brett[3][5] = 3;
10     brett[6][7] = 4;
11     // Ausgabe
12     int i, j;
13     // Schleife fuer Zeilen, Y-Achse
14     for (i=0; i<8; i++) {
15         // Schleife fuer Spalten, X-Achse
16         for (j=0; j<8; j++) {
17             std::cout << brett[i][j] << " ";
18         }
19         std::cout << "\n";
20     }
21     return 0;
22 }
```

```

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 3 0 0
0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 3 0 0
0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0

```

[1] Quelle: [C-Kurs](#)

## Anwendung eines zweidimensionalen Arrays

Elementweise Addition zweier Matrizen

### Addition.cpp

```

1  #include <iostream>
2
3  int main(void)
4  {
5      int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
6      int B[2][3] = {{10, 20, 30}, {40, 50, 60}};
7      int C[2][3];
8      int i, j;
9      for (i=0; i<2; i++)
10         for (j=0; j<3; j++)
11             C[i][j] = A[i][j] + B[i][j];
12     for (i=0; i<2; i++)
13     {
14         for (j=0; j<3; j++)
15             std::cout << C[i][j] << "\t";
16         std::cout << "\n";
17     }
18     return 0;
19 }

```



11	22	33
44	55	66
11	22	33
44	55	66

Weiteres Beispiel: Lösung eines Gleichungssystem mit dem Gausschen Eliminationsverfahren [Link](#)

**Merke:** Größere Daten in Arrays abzulegen ist in der Regel effizienter als einzelne Variablen zu verwenden. Die Verwendung von Arrays ist aber nicht immer die beste Lösung. Prüfen Sie höherabstraktere Formate wie Listen oder Vektoren!

## Sonderfall Zeichenketten / Strings

Folgen von Zeichen, die sogenannten *Strings* werden in C/C++ durch Arrays mit Elementen vom Datentyp `char` repräsentiert. Die Zeichenfolgen werden mit `\0` abgeschlossen.



```

1  #include <iostream>
2
3  int main(void) {
4      std::cout << "Diese Form eines Strings haben wir bereits mehrfach
        benutzt!\n";
5      ///////////////////////////////////////////////////////////////////
        //
6
7      char a[] = "Ich bin ein char Array!"; // Der Compiler fügt das \0
        automatisch ein!
8      if (a[23] == '\0') {
9          std::cout << "char Array Abschluss in a gefunden!\n";
10     }
11
12     std::cout << "->" << a << "<-\n";
13     char b[] = { 'H', 'a', 'l', 'l', 'o', ' ',
14                 'F', 'r', 'e', 'i', 'b', 'e', 'r', 'g', '\0' };
15     std::cout << "->" << b << "<-\n";
16     char c[] = "Noch eine \0Möglichkeit";
17     std::cout << "->" << c << "<-\n";
18     char d[] = { 69, 65, 86, 68, 32, 50, 48, 50, 52, 0 };
19     std::cout << "->" << d << "<-\n";
20
21     return 0;
22 }

```

```

Diese Form eines Strings haben wir bereits mehrfach benutzt!
char Array Abschluss in a gefunden!
->Ich bin ein char Array!<-
->Hallo Freiberg<-
->Noch eine <-
->EAVD 2024<-
Diese Form eines Strings haben wir bereits mehrfach benutzt!
char Array Abschluss in a gefunden!
->Ich bin ein char Array!<-
->Hallo Freiberg<-
->Noch eine <-
->EAVD 2024<-

```

C++ implementiert einen separaten string-Datentyp (Klasse), die einen deutlichen komfortableren Umgang mit Texten erlaubt. Beim Anlegen eines solchen muss nicht angegeben werden, wie viele Zeichen reserviert werden sollen. Zudem können Strings einfach zuweisen und verglichen werden, wie es für andere Datentypen üblich ist. Die C const char \* Mechanismen funktionieren aber auch hier.

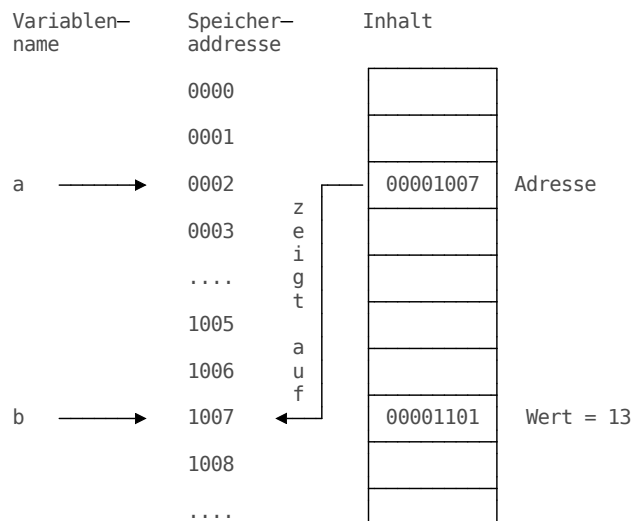
```
1 #include <iostream>
2 #include <string>      // Header für string Klasse
3
4 int main(void) {
5     std::string hanna = "Hanna";
6     std::string anna = "Anna";
7     std::string alleBeide = anna + " + " + hanna;
8     std::cout << "Hallo: " << alleBeide << "\n";
9
10    int res = anna.compare(hanna);
11    if (res == 0)
12        std::cout << "Both input strings are equal.\n";
13    else if (res < 0)
14        std::cout << "String 1 is smaller as compared to String 2\n.";
15    else
16        std::cout << "String 1 is greater as compared to String 2\n.";
17
18    return 0;
19 }
```

```
Hallo: Anna + Hanna
String 1 is smaller as compared to String 2
.Hallo: Anna + Hanna
String 1 is smaller as compared to String 2
.
```

## Grundkonzept Zeiger

Bisher umfassten unserer Variablen als Datencontainer Zahlen oder Buchstaben. Das Konzept des Zeigers (englisch Pointer) erweitert das Spektrum der Inhalte auf Adressen.

An dieser Adresse können entweder Daten, wie Variablen oder Objekte, aber auch Programmcodes (Anweisungen) stehen. Durch Dereferenzierung des Zeigers ist es möglich, auf die Daten oder den Code zuzugreifen.



Welche Vorteile ergeben sich aus der Nutzung von Zeigern, bzw. welche Programmiertechniken lassen sich realisieren:

- dynamische Verwaltung von Speicherbereichen,
- Übergabe von Datenobjekten an Funktionen via "call-by-reference",
- Übergabe von Funktionen als Argumente an andere Funktionen,
- Umsetzung rekursiver Datenstrukturen wie Listen und Bäume.

An dieser Stelle sei bereits erwähnt, dass die Übergabe der "call-by-reference"-Parameter via Reference ist ebenfalls möglich und einfacher in der Handhabung.

## Definition von Zeigern

Die Definition eines Zeigers besteht aus dem Datentyp des Zeigers und dem gewünschten Zeigernamen. Der Datentyp eines Zeigers besteht wiederum aus dem Datentyp des Werts auf den gezeigt wird sowie aus einem Asterisk. Ein Datentyp eines Zeigers wäre also z. B. `double*`.

```
/* kann eine Adresse aufnehmen, die auf einen Wert vom Typ Integer zeigt */
int* zeiger1;
/* das Leerzeichen kann sich vor oder nach dem Stern befinden */
float *zeiger2;
/* ebenfalls möglich */
char * zeiger3;
/* Definition von zwei Zeigern */
int *zeiger4, *zeiger5;
/* Definition eines Zeigers und einer Variablen vom Typ Integer */
int *zeiger6, ganzzahl;
```

# Initialisierung

**Merke:** Zeiger müssen vor der Verwendung initialisiert werden.

Der Zeiger kann initialisiert werden durch die Zuweisung:

- der Adresse einer Variable, wobei die Adresse mit Hilfe des Adressoperators `&` ermittelt wird,
- eines Arrays,
- eines weiteren Zeigers oder
- des Wertes von `NULL`.

<!-- TODO: Sollten wir im C++ Kontext nicht mind. auf `nullptr` verweisen? -->

## PointerExamples.cpp

```
1  #include <iostream>
2
3  int main(void) {
4      int a = 0;
5      int * ptr_a = &a;          /* mit Adressoperator */
6
7      int feld[10];
8      int * ptr_feld = feld;     /* mit Array */
9
10     int * ptr_b = ptr_a;       /* mit weiterem Zeiger */
11
12     int * ptr_Null = NULL;     /* mit NULL */
13
14     std::cout << "Pointer ptr_a      " << ptr_a << "\n";
15     std::cout << "Pointer ptr_feld  " << ptr_feld << "\n";
16     std::cout << "Pointer ptr_b      " << ptr_b << "\n";
17     std::cout << "Pointer ptr_Null " << ptr_Null << "\n";
18     return 0;
19 }
```

```
Pointer ptr_a      0x7fff713895ec
Pointer ptr_feld 0x7fff71389610
Pointer ptr_b      0x7fff713895ec
Pointer ptr_Null 0
Pointer ptr_a      0x7fff172bc69c
Pointer ptr_feld 0x7fff172bc6c0
Pointer ptr_b      0x7fff172bc69c
Pointer ptr_Null 0
```

Die konkrete Zuordnung einer Variablen im Speicher wird durch den Compiler und das Betriebssystem bestimmt. Entsprechend kann die Adresse einer Variablen nicht durch den Programmierer festgelegt werden. Ohne Manipulationen ist die Adresse einer Variablen über die gesamte Laufzeit des Programms unveränderlich, ist aber bei mehrmaligen Programmstarts unterschiedlich.

In den Ausgaben von Pointer wird dann eine hexadezimale Adresse ausgegeben.

Zeiger können mit dem "Wert" `NULL` als ungültig markiert werden. Eine Dereferenzierung führt dann meistens zu einem Laufzeitfehler nebst Programmabbruch. `NULL` ist ein Macro und wird in mehreren Header-Dateien definiert (mindestens in `<cstdlib>` (`stddef.h`)). Die Definition ist vom Standard implementierungsabhängig vorgegeben und vom Compilerhersteller passend implementiert, z. B.

```
#define NULL 0
#define NULL 0L
#define NULL (void *) 0
```

Und umgekehrt, wie erhalten wir den Wert, auf den der Pointer zeigt? Hierfür benötigen wir den *Inhaltsoperator* `*`.

#### DereferencingPointers.cpp

```
1  #include <iostream>
2
3  int main(void) {
4      int a = 15;
5      int * ptr_a = &a;
6      std::cout << "Wert von a" << a << "\n";
7      std::cout << "Pointer ptr_a" << ptr_a << "\n";
8      std::cout << "Wert hinter dem Pointer ptr_a" << *ptr_a << "\n";
9      *ptr_a = 10;
10     std::cout << "Wert von a" << a << "\n";
11     std::cout << "Wert hinter dem Pointer ptr_a" << *ptr_a << "\n";
12     return 0;
13 }
```

```
Wert von a          15
Pointer ptr_a       0x7fff8a9306cc
Wert hinter dem Pointer ptr_a 15
Wert von a          10
Wert hinter dem Pointer ptr_a 10
Wert von a          15
Pointer ptr_a       0x7ffe94641b7c
Wert hinter dem Pointer ptr_a 15
Wert von a          10
Wert hinter dem Pointer ptr_a 10
```

Schauen wir wiederum auf eine grafische Darstellung [PythonTutor](#)

## Fehlerquellen

Fehlender Adressoperator bei der Zuweisung

### PointerFailuresI.cpp

```
1  #include <iostream>
2
3  int main(void) {
4      int a = 5;
5      int * ptr_a;
6      ptr_a = a;
7      std::cout << "Pointer ptr_a          " << ptr_a << "\n";
8      std::cout << "Wert hinter dem Pointer ptr_a " << *ptr_a << "\n";
9      std::cout << "Aus Maus!\n";
10     return 0;
11 }
```

main.cpp: In function 'int main()':

main.cpp:6:11: error: invalid conversion from 'int' to 'int\*' [-fpermissive]

```
6 |   ptr_a = a;
  |           ^
  |           |
  |           int
```

main.cpp: In function 'int main()':

main.cpp:6:11: error: invalid conversion from 'int' to 'int\*' [-fpermissive]

```
6 |   ptr_a = a;
  |           ^
  |           |
  |           int
```

Fehlender Dereferenzierungsoperator beim Zugriff

## PointerFailuresII.cpp



```
1  #include <iostream>
2
3  int main(void)
4  {
5      int a = 5;
6      int * ptr_a = &a;
7      std::cout << "Pointer ptr_a          " << ptr_a << "\n";
8      std::cout << "Wert hinter dem Pointer ptr_a  " << ptr_a << "\n";
9      std::cout << "Aus Maus!\n";
10     return 0;
11 }
```

```
Pointer ptr_a          0x7ffcd534c10c
Wert hinter dem Pointer ptr_a  0x7ffcd534c10c
Aus Maus!
Pointer ptr_a          0x7ffd052d818c
Wert hinter dem Pointer ptr_a  0x7ffd052d818c
Aus Maus!
```

Uninitialisierte Pointer zeigen "irgendwo ins nirgendwo"!

## PointerFailuresIII.cpp



```
1  #include <iostream>
2
3  int main(void) {
4      int * ptr_a;
5      // korrekte Initialisierung
6      // int * ptr_a = NULL;
7
8      // ... hier passiert irgendwas
9
10     *ptr_a = 10;
11     // Prüfung auf gültige Adresse
12     // if (ptr_a != NULL) *ptr_a = 10;
13     std::cout << "Pointer ptr_a          " << ptr_a << "\n";
14     std::cout << "Wert hinter dem Pointer ptr_a  " << *ptr_a << "\n";
15     std::cout << "Aus Maus!\n";
16     return 0;
17 }
```



```

main.cpp: In function 'int main()':
main.cpp:10:10: warning: 'ptr_a' is used uninitialized [-Wuninitialized]
    10 |     *ptr_a = 10;
        |     ~~~~~^~~~

main.cpp: In function 'int main()':
main.cpp:10:10: warning: 'ptr_a' is used uninitialized [-Wuninitialized]
    10 |     *ptr_a = 10;
        |     ~~~~~^~~~

```

## Dynamische Datenobjekte

C++ bietet die Möglichkeit den Speicherplatz für eine Variable zur Laufzeit zur Verfügung zu stellen. Mit dem `new`-Operator wird der Speicherplatz bereit gestellt und mit dem `delete`-Operator (`delete[]`) wieder freigegeben.

`new` erkennt die benötigte Speichermenge am angegebenen Datentyp und reserviert für die Variable auf dem Heap die entsprechende Byte-Menge.

new.cpp

```

1  #include <iostream>
2
3  int main(void) {
4      int * ptr_a;
5      ptr_a = new int;
6      *ptr_a = 10;
7      std::cout << "Pointer ptr_a          " << ptr_a << "\n";
8      std::cout << "Wert hinter dem Pointer ptr_a  " << *ptr_a << "\n";
9      std::cout << "Aus Maus!\n";
10     delete ptr_a;
11     return 0;
12 }

```

```

Pointer ptr_a          0x56460a4dbeb0
Wert hinter dem Pointer ptr_a  10
Aus Maus!
Pointer ptr_a          0x560ccaea9eb0
Wert hinter dem Pointer ptr_a  10
Aus Maus!

```



```
1  #include <iostream>
2
3  int main(void) {
4      int * ptr_a;
5      ptr_a = new int[3];
6      ptr_a[0] = ptr_a[1] = ptr_a[2] = 42;
7      std::cout << "Werte hinter dem Pointer ptr_a:  ";
8      for (int i=0;i<3;i++)
9          std::cout << ptr_a[i] << " ";
10     std::cout << "\n";
11     std::cout << "Aus Maus!\n";
12     delete[] ptr_a;
13     return 0;
14 }
```

```
Werte hinter dem Pointer ptr_a:  42 42 42
Aus Maus!
Werte hinter dem Pointer ptr_a:  42 42 42
Aus Maus!
```

- `delete` darf nur einmal auf ein Objekt angewendet werden
- `delete` darf ausschließlich auf mit `new` angelegte Objekte oder NULL-Pointer angewandt werden
- Nach der Verwendung von `delete` ist das Objekt *undefiniert* (nicht gleich NULL)

**Merke:** Die Verwendung von Zeigern kann zur unerwünschten Speicherfragmentierung und die Programmierfehler zu den Programmabstürzen und Speicherlecks führen. *Intelligente* Zeiger stellen sicher, dass Programme frei von Arbeitsspeicher- und Ressourcenverlusten sind.

## Referenz

Eine Referenz ist eine Datentyp, der Verweis (Aliasnamen) auf ein Objekt liefert und ist genau wie eine Variable zu benutzen ist. Der Vorteil der Referenzen gegenüber den Zeigern besteht in der einfachen Nutzung:

- Dereferenzierung ist nicht notwendig, der Compiler löst die Referenz selbst auf
- Freigabe ist ebenfalls nicht notwendig

**Merke:** Auch Referenzen müssen vor der Verwendung initialisiert werden. Eine Referenz bezieht sich immer auf ein existierendes Objekt, sie kann nie NULL sein

#### referenzen.cpp



```
1 #include <iostream>
2
3 int main(void) {
4     int a = 1; // Variable
5     int &r = a; // Referenz auf die Variable a
6
7     std::cout << "a: " << a << " r: " << r << std::endl;
8     std::cout << "a: " << &a << " r: " << &r << std::endl;
9 }
```

```
a: 1 r: 1
a: 0x7fffb4dd8c4c r: 0x7fffb4dd8c4c
```

Die Referenzen werden verwendet:

- zur "call by reference"-Parameterübergabe
- zur Optimierung des Programms, um Kopien von Objekten zu vermeiden
- in speziellen Memberfunktionen, wie Copy-Konstruktor und Zuweisungsoperator
- als sogenannte universelle Referenz (engl.: universal reference), die bei Templates einen beliebigen Parametertyp repräsentiert.

**Achtung:** Zur dynamischen Verwaltung von Speicherbereichen sind Referenzen nicht geeignet.

## Beispiel der Woche

Gegeben ist ein Array, das eine sortierte Reihung von Ganzzahlen umfasst. Geben Sie alle Paare von Einträgen zurück, die in der Summe 18 ergeben.

Die intuitive Lösung entwirft einen kreuzweisen Vergleich aller sinnvollen Kombinationen der  $n$  Einträge im Array. Dafür müssen wir  $(n - 1)^2 / 2$  Kombinationen bilden.

	<b>1</b>	<b>2</b>	<b>5</b>	<b>7</b>	<b>9</b>	<b>10</b>	<b>12</b>	<b>13</b>	<b>16</b>	<b>17</b>	<b>18</b>	<b>21</b>	<b>25</b>
1	x									18			
2	x	x							18				
5	x	x	x					18					
7	x	x	x	x									
9	x	x	x	x	x								
10	x	x	x	x	x	x							
12	x	x	x	x	x	x	x						
13	x	x	x	x	x	x	x	x					
16	x	x	x	x	x	x	x	x	x				
17	x	x	x	x	x	x	x	x	x	x			
18	x	x	x	x	x	x	x	x	x	x	x		
21	x	x	x	x	x	x	x	x	x	x	x	x	
25	x	x	x	x	x	x	x	x	x	x	x	x	x

Haben Sie eine bessere Idee?



```
1  #include <iostream>
2  #define ZIELWERT 18
3
4  int main(void) {
5      int a[] = {1, 2, 5, 7, 9, 10, 12, 13, 16, 17, 18, 21, 25};
6      int i_left = 0;
7      int i_right = 12;
8
9      std::cout << "Value left " << a[i_left] << " right " << a[i_right]
10     -----\n";
11     do {
12         std::cout << "Value left " << a[i_left] << " right " << a[i_right]
13         if (a[i_right] + a[i_left] == ZIELWERT)
14             std::cout << " -> TREFFER";
15         std::cout << "\n";
16
17         if (a[i_right] + a[i_left] >= ZIELWERT)
18             i_right--;
19         else
20             i_left++;
21     } while (i_right != i_left);
22     return 0;
23 }
```

```
Value left 1 right 25
-----
Value left 1 right 25
Value left 1 right 21
Value left 1 right 18
Value left 1 right 17 -> TREFFER
Value left 1 right 16
Value left 2 right 16 -> TREFFER
Value left 2 right 13
Value left 5 right 13 -> TREFFER
Value left 5 right 12
Value left 7 right 12
Value left 7 right 10
Value left 9 right 10
Value left 1 right 25
-----
Value left 1 right 25
Value left 1 right 21
Value left 1 right 18
Value left 1 right 17 -> TREFFER
Value left 1 right 16
Value left 2 right 16 -> TREFFER
Value left 2 right 13
Value left 5 right 13 -> TREFFER
Value left 5 right 12
Value left 7 right 12
Value left 7 right 10
Value left 9 right 10
```

## Quiz

### Arrays

Erstellen Sie ein eindimensionales Array namens `arr`, das 7 Elemente vom Typ `int` enthält.

Erstellen Sie ein zweidimensionales Array namens `arr`, das 3\*4 Elemente vom Typ `int` enthält.

## Zugriff

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main(void) {
    float b[5] = {1.0, 4.8, 1.2, 42.0, 99.0};
    std::cout << b[2] << "\n";
    return 0;
}
```



Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
    int a[5] = {5, 8};
    std::cout << a[2] << "\n";
    return 0;
}
```



## Mehrdimensionale Arrays

Es existiert ein Array `int A[2][5];`. Setzen Sie `[_____]` gleich 1.

	Spalten			
Zeilen	<code>a[0][0]</code>	<code>a[0][1]</code>	...	
			<code>[_____]</code>	

Durch was muss `[_____]` ersetzt werden damit die Zahl `19` aus `m[4][5]` ausgegeben wird?

```
#include <iostream>

int main() {
    int m[4][5] = {{1, 2, 3, 4, 5},
                  {6, 7, 8, 9, 10},
                  {11, 12, 13, 14, 15},
                  {16, 17, 18, 19, 20}};
    std::cout << [_____] << "\n";
    return 0;
}
```

?

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
    int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int B[2][3] = {{10, 20, 30}, {40, 50, 60}};

    std::cout << A[1][0] + B[0][1] << "\n";
    return 0;
}
```



## Zeichenketten

Durch welche Sequenz werden Zeichenketten abgeschlossen?

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
    char c[] = "Peter wohnt irgendwo\0 in Freiberg.";
    std::cout << c << "\n";
}
```



## Zeiger

Worauf zeigen Zeiger?

- ☒ chars
- ☐ Referenzen
- ☐ Speicheradressen

## Definition

Welche der folgenden Definitionen sind möglich?

- ☐ `int* z1;`
- ☐ `float * z2;`
- ☐ `char *z3;`
- ☐ `int *z4, *z5;`
- ☐ `int z6*;`
- ☐ `int *z7, i;`

## Initialisierung

Durch welches Zeichen werden Adressen ermittelt?

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
    int a = 15;
    int *ptr_a = &a;
    std::cout << *ptr_a << "\n";

    return 0;
}
```



- ☐ Die Adresse von `a`
- ☐ 15
- ☐ `NULL`

## Dynamische Datenobjekte

Wie häufig kann `delete` auf ein Objekt angewendet werden?

- ☐ 0
- ☐ 1
- ☐ 42
- ☐ Beliebig oft

Wie lautet die Aussage dieses Programms?

```
#include <iostream>

int main() {
    int a = 10;
    int *ptr_a = &a;
    std::cout << ptr_a << "\n";
    delete ptr_a;
    return 0;
}
```

- ☐ 10
- ☐ Die Adresse von `a`
- ☐ Die Adresse des Zeigers `*ptr_a`
- ☐ Es gibt einen Error

## Referenz

Welche der im Beispiel benutzten Variablen ist eine Referenz?

```
#include <iostream>

int main() {
    int a = 10;
    int &b = a;
    int *c = b;
    std::cout << c << "\n";
    return 0;
}
```

- ☐ a
- ☐ b
- ☐ c

Hier ist ein Programm mit Ausgabe vorgegeben. Was müsste statt [\_\_\_\_\_] ausgegeben werden?

```
#include <iostream>

int main(void) {
    int a = 1;
    int &r = a;

    std::cout << "a: " << &a << " r: " << &r << endl;
}
```

a: [\_\_\_\_\_] r: 0x7ffddddd212fc

Hier ist ein Programm mit Ausgabe vorgegeben. Was müsste statt [\_\_\_\_\_] ausgegeben werden?

```
#include <iostream>

int main(void) {
    int a = 1;
    int &r = a;

    std::cout << "a: " << a << " r: " << r << "\n";
}
```

a: 1 r: [\_\_\_\_\_]