Zeiger

- Zeiger und Adressen

- C-Arrays
  - C-Arrays und `sizeof`
  - Indexoperator bei C-Arrays
  - Initialisierung von C-Arrays
  - Zeigerarithmetik
  - Mehrdimensionale Felder

- C-Zeichenketten

- Zeiger und Funktionen
  - Parameter der Funktion `main()`
  - Parameterübergabe mit Zeigern
  - Gefahren bei der Rückgabe von Zeigern
  - Zeiger auf Funktionen

- Dynamische Datenobjekte
  - Freigeben dynamischer Objekte
  - Die Funktion `set_new_handler()`
  - Dynamisch erzeugte Strukturen
  - Speicherverwaltung in C vs. C++

Zeiger

Zeiger und Adressen

Zeiger sind ähnlich wie andere Variablen: Sie haben einen Namen, einen Wert, und sie können mit Operatoren verändert werden. Der Unterschied besteht darin, dass der Wert als *Adresse* behandelt wird.

In *Deklarationen* bedeutet ein `*` „Zeiger auf“:

```
int *ip;
```

`ip` ist ein Zeiger auf einen `int`-Wert. In der Speicherzelle, auf die `ip` zeigt, befindet sich ein `int`-Wert.

In *Anweisungen* bedeutet `*` eine *Dereferenzierung*:

```
*ip = 100;
```

setzt den Wert der Speicherzelle, auf die `ip` zeigt, auf 100.

Zeiger

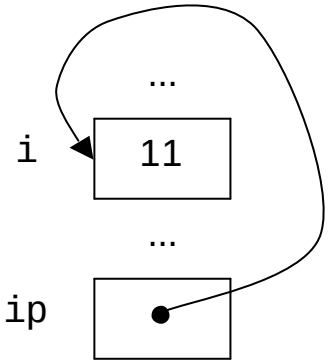
Zeiger und Adressen

Beispiele:

```
int i = 11;
int *ip;
ip = &i;        // Adressoperator &
```

...	Adresse	Name
	121	
	122	
11	**123**	**i**
	124	
...	125	

vereinfacht:



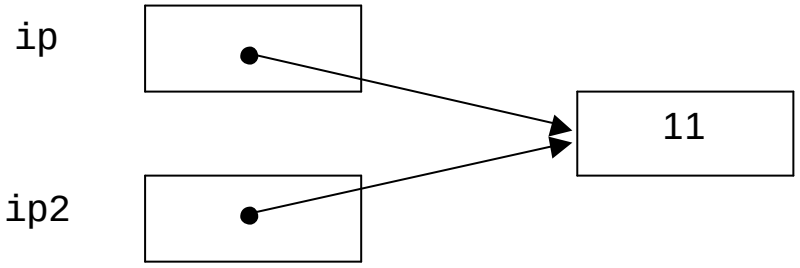
...	Adresse	Name
	221	
	222	
	223	
123	**224**	**ip**
...	225	

Zeiger

Zeiger und Adressen

Beispiele:

```
int *ip2 = &i;
```



Die Anweisungen

```
  i  = 99;
*ip  = 99;
*ip2 = 99;
```

bewirken jetzt alle dasselbe. `*ip` und `*ip2` sind *Aliasnamen* für `i`.

Zeiger

Zeiger und Adressen

Schreibweise von Zeigerdeklarationen

Die folgenden drei Schreibweisen sind äquivalent:

```
// x ist vom Typ int,
// ip ist vom Typ "Zeiger auf int"
int*  ip, x;
int  *  ip, x;
int *ip, x;

// nur eine Variable pro Deklaration:
// Verwechslung nicht möglich
int *ip;
int x;

int* ip;
int x;
```

Zeiger

Zeiger und Adressen

NULL-Zeiger

Ein mit `NULL` initialisierter Zeiger zeigt nicht irgendwohin, sondern definitiv auf „nichts“.

Ein Zeiger kann auf den Wert `NULL` geprüft werden:

```
int * ip;
...
if(ip != NULL)
   ...
```

Die Definition von `NULL` ist im Header `<cstddef>` enthalten.

Zeiger

Zeiger und Adressen

Typprüfung

Im Gegensatz zu C wird der Typ eines Zeigers in C++ geprüft.

```
char *cp;      // Zeiger auf char
void *vp;      // Zeiger auf void
```

`void` hat die Bedeutung: beliebiger Datentyp. vp ist also ein Zeiger auf ein Objekt eines beliebigen Datentyps.

```
vp = cp;       // möglich
cp = vp;       // nicht möglich,
               // Fehlermeldung des Compilers
```

Die Zuweisung kann durch explizite Typumwandlung ermöglicht werden.

```
cp = static_cast<char*>(vp);
```

Zeiger

Zeiger und Adressen

Zeigerzugriff auf automatische Variablen

Der Speicherplatz für die innerhalb eines Blocks deklarierten Variablen wird bei Verlassen des Blocks wieder freigegeben. Der nachträgliche Zugriff auf diese Variablen kann zu Dateninkonsistenzen und zum Systemabsturz führen.

Beispiel:

```
int i = 9;
int *ip = &i;
*ip = 77;

{
   int j = 888;
   ip = &j;
   ...
}           // Blockende, j wird ungueltig

*ip = 77;   // Das kann ins Auge gehen.
```

Zeiger

Zeiger und Adressen

Konstante Zeiger auf konstante Werte

In den Deklarationen

```
const char *buf;
char const *buf;
```

ist `buf` ein Zeiger auf konstante `chars`. Also ist

```
buf++;          // erlaubt
*buf = 'a';     // verboten
```

In der Deklaration

```
char * const buf;
```

ist `buf` ein konstanter Zeiger, der nicht verändert werden darf. Das, worauf `buf` zeigt, darf verändert werden.

Auch die Deklarationen

```
const char * const buf;
char const * const buf;
```

sind möglich. Hier darf weder der Zeiger noch das, worauf er verweist, verändert werden.

Zeiger

Zeiger und Adressen

Unterschiede zwischen Zeigern und Referenzen

Eine **Referenz** ist ein Datentyp, der einen *Verweis* auf ein Objekt liefert.

Eine Referenz bildet einen *Aliasnamen* für ein Objekt, über den es ansprechbar ist. Das Objekt hat damit zwei Namen.

Beispiel:

```
int intValue;
int &ref = intValue;
```

Die folgenden beiden Anweisungen haben dann den gleichen Effekt:

```
intValue++;
ref++;
```

Referenzen müssen bei der Deklaration initialisiert werden:

```
int &ref;
```

ist nicht erlaubt.

Eine Referenz auf eine Referenz ist nicht möglich, und auch ein Zeiger auf eine Referenz ist nicht erlaubt.

Zeiger

Zeiger und Adressen

Unterschiede zwischen Zeigern und Referenzen

Referenzen können syntaktisch wie ein Objektname verwendet werden. Der Compiler löst die Referenz auf, während beim Zeiger stets vom Programmierer dereferenziert werden muss (mit dem Inhaltsoperator *).

Eine Referenz muss initialisiert werden.Sobald sie initialisiert ist, kann sie nicht mehr für ein anderes Objekt verwendet werden.

Eine Referenz bezieht sich immer auf ein existierendes Objekt, sie kann nie NULL sein. Eine Deklaration wie z.B.

```
int &ref;
```

ist nicht erlaubt. Zeiger sind eigenständige Variablen; sie „zeigen“ entweder auf eine konkrete Adresse oder auf NULL.

Wenn der Adress-Operator & auf eine Referenz angewandt wird, liefert dieser Ausdruck die Adresse der referenzierten Variablen. Die Adresse eines Zeigers hat nichts mit der Adresse der Variablen, auf die er zeigt, zu tun.

Zeiger

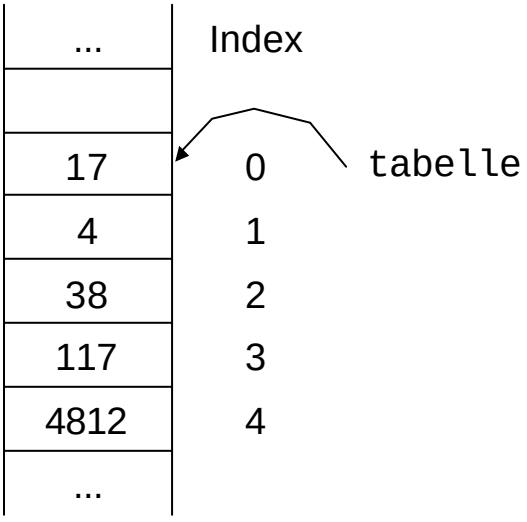
C-Arrays

Ein *C-Array* genanntes Feld *ist eine Tabelle von Elementen desselben Datentyps*. Die Vektoren der C++-Standardbibliothek basieren auf diesen Feldern. Die Definition eines Feldes lautet:

Datentyp Feldname [*Anzahl_der_Elemente*];

Anzahl_der_Elemente muss eine Konstante sein oder ein Ausdruck, der ein konstantes Ergebnis hat:

```
const int anzahl = 5;
int tabelle[anzahl];
```



Der *Name des Feldes* zeigt auf die *Anfangsadresse* des Feldes, d.h. auf das erste Feldelement und ist auch wie ein Zeiger verwendbar.

Zeiger

C-Arrays

Der Zugriff auf ein Feldelement geschieht durch den Indexoperator `[]` oder durch Zeigerarithmetik.

Beispiel:

```
char a[100];
a[5]     = '?';  // gleichwertig:
*(a + 5) = '?';  // Wert an der Stelle a + 5

char c = a[9];

char* cp;      // Zeiger auf char
cp = &c;       // erlaubt
cp = a;        // cp zeigt auf den Feldanfang,
               // d.h. *cp == a[0]
a = cp;        // Fehler: a ist kein L-Wert
a = &c;        // Fehler: a ist kein L-Wert
```

Zeiger

C-Arrays

Unterschiede zwischen Zeigern und Arrays

- Ein Zeiger hat einen Speicherplatz, der einen Wert enthält, der als Speicheradresse benutzt werden kann.
- Ein Array ist ein symbolischer Name für eine Adresse (= den Anfang) eines Bereichs im Speicher. Der Name wird syntaktisch wie ein konstanter Zeiger behandelt.

C-Arrays

C-Arrays und `sizeof`

C-Arrays sind keine Objekte im bisherigen Sinne. Es gibt keine Methoden, die verwendet werden können:

```
// Fuer ein Array wie z.B.
double kosten[31];
// gibt es keine Methode size()
// => Fehlermeldung des Compilers:
for(int i = 0; i < kosten.size(); i++)
   cout << i << " : " << kosten[i] << endl;
```

Die Größe eines Arrays kann nicht von ihm erfragt werden, sie muss vorher bekannt oder mit `sizeof` ermittelt worden sein:

```
const int anzahl = 5;
int tabelle[anzahl];
// ...
for(int i = 0; i < anzahl; i++)
   cout << i << " : " << tabelle[i] << endl;
for(int i = 0;
    i < sizeof(tabelle) / sizeof(tabelle[0]);
    i++)
   cout << i << " : " << tabelle[i] << endl;
```

`sizeof()` ist eine Funktion, die den Platzbedarf des in Klammern stehenden Objekts in Bytes zurückgibt.

C-Arrays

C-Arrays und `sizeof`

`sizeof` funktioniert nicht bei C-Arrays, die als Parameter einer Funktion übergeben werden, weil C-Arrays innerhalb der Funktion nur als Zeiger interpretiert werden:

```
#include <iostream>
using namespace std;

void tabellenAusgabe(int tabelle[])
// in der Parameterliste nur C-Array-Deklaration
{
   int anzahlBytes = sizeof(tabelle);  // Fehler!
   int wieOft = anzahlBytes / sizeof(int);

   for(int i = 0; i < wieOft; i++)
      cout << i << " : " << tabelle[i] << endl;
}

int main()
{
   const int anzahl = 5;
   int tabelle[anzahl];       // C-Array-Definition
   // ...
   tabellenAusgabe(tabelle);  // leider falsch
   return 0;
}
```

`sizeof` gibt nur den Platzbedarf für einen Zeiger zurück, weil syntaktisch

`int tabelle[]`	dasselbe wie
`int* tabelle`	ist.

C-Arrays

Indexoperator bei C-Arrays

Der Zugriff mit dem Index-Operator wird nicht auf Bereichsüberschreitung überprüft.

Man kann zum Zugriff auf ein Feldelement auch die Zeiger-Notation verwenden, also z.B. statt

```
tabelle[i]        auch
*(tabelle + i)    schreiben.
```

Initialisierung von C-Arrays

Ein C-Array kann bei der Definition bereits initialisiert werden.

```
int feld[] = {1, 777, 888};
const int anzahl = sizeof(feld) /
                   sizeof(feld[0]);
```

Falls die Initialisierungsliste weniger Werte enthält als Array-Elemente vorhanden sind, werden die restlichen Elemente mit 0 initialisiert.

```
int feld[3] = {1};
```

ist identisch mit

```
int feld[3] = {1, 0, 0};
```

C-Arrays

Zeigerarithmetik

Wenn ein Zeiger inkrementiert oder dekrementiert wird, zeigt er nicht auf die nächste Speicheradresse, sondern auf die Adresse des nächsten Werts. Der Abstand der Speicheradressen ist gleich der Größe, die der Wert beansprucht, z.B. 8 beim Datentyp `double`.

Beispiele:

```
double d[10];
double *dp1 = d;
double *dp2 = dp1 + 1;

cout << dp2 – dp1 << endl;        // 1
cout << long(dp2) – long(dp1);    // 8
```

dp2 zeigt auf das nächste `double`-Element des Arrays d.

```
char c[10];
char *cp1 = c;
char *cp2 = cp1 + 1;

cout << cp2 – cp1 << endl;        // 1
cout << long(cp2) – long(cp1);    // 1
```

C-Arrays

Zeigerarithmetik

Beispiel: Suche eines Werts in einer Tabelle

Es wird vorausgesetzt, dass das Feld um einen Eintrag erweitert wird, der als „Wächter“ (englisch *sentinel*) für den Abbruch der Schleife dient.

```
// Definitionen
const int n = ...
int a[n + 1];       // C-Array
int key = ...       // gesuchtes Element
int i;              // Laufvariable

// Ergebnis: i == 0 ... n-1 : gefunden
//           i == n : nicht gefunden

a[n] = key;
int *p = a;
while(*p++ != key)
   ;
i = p – a – 1;    // Zeigerarithmetik
```

C-Arrays

Mehrdimensionale Felder

Mehrdimensionale Felder sind Felder, deren Elemente selbst wieder Felder sind. So sind die Zeilen des folgenden zweidimensionalen Feldes `matrix` eindimensionale Felder:

```
#include<iostream>
using namespace std;

int main()
{
    const int dim1 = 2, dim2 = 3;
    int matrix [dim1] [dim2] = {{1,2,3}, {4,5,6}};
    // folgende Initialisierung ist aequivalent:
    // int matrix [dim1] [dim2] = {1,2,3,4,5,6};

    for (int i = 0; i < dim1; ++i)
    {
        for (int j = 0; j < dim2; ++j)
            cout << matrix[i][j] << ' ';
        cout << endl;
    }
}
```

Ausgabe:

```
1 2 3
4 5 6
```

C-Arrays

Mehrdimensionale Felder

Mehrdimensionale Arrays haben für jede Dimension ein Klammerpaar `[]`. Die folgenden Schreibweisen sind äquivalent:

```
matrix[i][j]
*(matrix[i] + j)
*(*(matrix + i) + j)
```

`matrix[i]` können wir als Zeiger auf die `i`-te Zeile des Feldes interpretieren.

Quiz:

Was ist der Unterschied zwischen den beiden folgenden Array-Zugriffen:

```
matrix[i][j]
matrix[i, j]
```

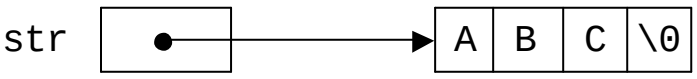
Zeiger

C-Zeichenketten

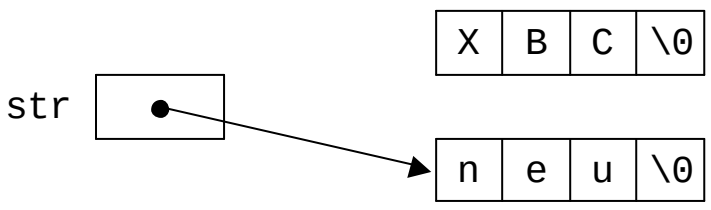
Eine *C-Zeichenkette* (englisch *string*) ist ein Spezialfall eines Arrays. Mit C-String ist eine Folge von Zeichen des Typs `char` gemeint, die mit einem NULL-Byte `'\0'` abgeschlossen wird.

Der Compiler erkennt einen C-String am Datentyp `char*`. Ein Zeichenketten-Literal erkennt der Compiler daran, dass es in Anführungszeichen eingeschlossen ist.

```
char * str = "ABC";
cout << str;                // ABC
int i = 1;
cout << str[i];             // B
cout << *(str + i);         // B
```




```
str[0] = 'X';    // gibt evtl. Laufzeitfehler
cout << str;                // XBC
str = "neu";
```



Zeiger

C-Zeichenketten

Zur Bearbeitung von C-Strings gibt es eine Reihe vordefinierter Funktionen, die in der Datei *string.h* (Header <cstring>) deklariert sind. Wegen der C++-Standardklasse `string` haben diese Funktionen stark an Bedeutung verloren.

Beispiel: die Funktion `strlen()`

```
#include <cstring>
#include <iostream>
using namespace std;

int main()
{
    const char* text =
    "Bei Initialisierung, Zuweisung, oder "
    "Ausgabe kann ein Stringliteral wie "
    "hier auch aus einzelnen Teilstrings "
    "zusammengesetzt werden.";

    cout << text << endl
         << " enthaelt " << strlen(text)
         << " Zeichen." << endl;

    return 0;
}
```

Zeiger

C-Zeichenketten

Ein nicht initialisierter C-String besitzt keinen Speicherplatz außer für den Zeiger selbst.

Bei der Eingabe mit `cin` muss darauf geachtet werden, dass genügend Platz zur Verfügung steht.

```
char* x;
cin >> x;                    // Fehler!

x = "1234567890";
cin >> x;                    // gefaehrlich!

char bereich[256];
// relativ sicher, liest bis zum ersten
// Zwischenraumzeichen
cin >> bereich;

const int zmax = 256;
char zeile[zmax];
// sicher, liest eine Zeile,
// aber maximal (zmax – 1) Zeichen
cin.getline(zeile, zmax);
```

Zeiger

C-Zeichenketten

char-Arrays

```
// Definition und Initialisierung
char str_a[ ] = "noch ein String";
char str_b[9];
char str_c[9] = "ABC";
cin >> str_b;

// Aliasing, d.h. mehr als ein Name fuer eine Sache
char* str1 = "Guten Morgen!\n";
char* str2 = str1;
cout << str2;

// erlaubte Zuweisung oder Aenderung
char charArray1[] = "hallo";
str1 = charArray1;
str1++;

// nicht erlaubte Aenderungen,
// weil ein Array kein L-Wert ist
charArray1 = str1;                    // Fehler!
charArray1++;                         // Fehler!

// Fehler! - kein Platz fuer '\0':
char buchstaben[3] = "abc";
//tolerante Compiler ignorieren die 3; besser ist:
char buchstaben[4] = "abc";
// Compiler zaehlen lassen
char buchstaben[ ] = "abc";
// ohne '\0'-Terminierung
char buchstaben[3] = {'a', 'b', 'c'};
```

Zeiger

C-Zeichenketten

Beispiele: Länge eines C-String berechnen

Version 1:

```
char *str = "Wie man sich bettet, "
            "so schallt es heraus.";
char *tmp = str;
int len = 0;
while(*tmp)
{
   len++;
   tmp++;
}
cout << "Laenge des Strings " << str
     << " = " << len << endl;
```

Version 2:

```
char *str = "Lieber reich und gesund "
            "als arm und krank.";
char *tmp = str;
int len = 0;
while(*tmp++)
   len++;
```

Zeiger

C-Zeichenketten

Beispiele: Länge eines C-String berechnen

Version 3:

```
char *str = "Morgenstund ist aller Laster Anfang.";
int len = 0;
while(str[len++])
   ;
--len;
```

Version 4:

```
char *str = "Aller Laster Anfang "
            "ist die Stossstange.";
int len = -1;
while(str[++len])
   ;
```

Version 5:

```
char *str = "Lieber Arm dran, als Arm ab.";
char *tmp = str;
while(*tmp++)
   ;
int len = tmp – str - 1;
```

Zeiger

C-Zeichenketten

Beispiele: C-Strings kopieren

Zwei C-Strings wurden deklariert und der zweite soll ein Duplikat des ersten werden:

```
char *original = "Ich verstehe mich! "
                 "(G. Ch. Lichtenberg)";
char * duplikat;
duplikat = original;
```

Es wird hier nur der Zeiger kopiert.
`duplikat` zeigt auf denselben Speicherbereich wie `original`.

Wenn eine C-Zeichenkette dupliziert werden soll, muss man sie elementweise in einen bereits definierten Speicherbereich kopieren.
Der `duplikat` zur Verfügung stehende Speicherplatz muss mindestens so groß sein, wie der von `original`.

Die folgenden Definitionen seien jeweils gegeben:

```
char* source = "Wer Anderen in der Nase bohrt, "
               "... ";
char dest[80];
```

Zeiger

C-Zeichenketten

Beispiele: C-Strings kopieren

Version 1:

```
int i = 0;
while(source[i] != '\0')
{
   dest[i] = source[i];
   i++;
}
dest[i] = '\0';
```

Version 2:

```
int i = -1;
while(source[++i])
   dest[i] = source[i];
dest[i] = '\0';
```

Version 3:

```
char *s = source, *d = dest;
while(*s)
{
   *d = *s;
   s++;
   d++;
}
*d = '\0';
```

Zeiger

C-Zeichenketten

Beispiele: C-Strings kopieren

Version 4:

```
char *s = source, *d = dest;
while(*d++ = *s++)
   ;
```

`while(*d++ = *s++)`
`   ;`
können wir als Abkürzung einer do-`while`-Schleife auffassen:

```
bool test;
do
{
   *d = *s;
   test = (*d != '\0');
   d++;
   s++;
} while(test);
```

Zeiger

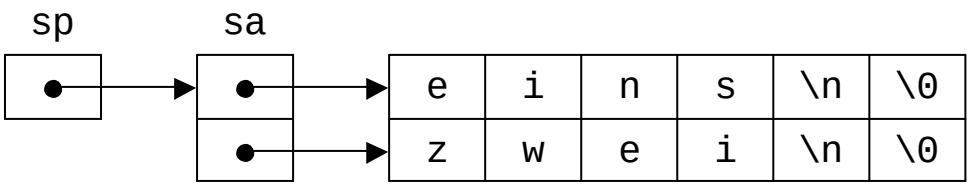
C-Zeichenketten

C-String-Arrays

Die Elemente eines Arrays können auch C-Strings sein.

Beispiel:

```
#include<iostream>
using namespace std;

int main()
{
   char* sa[] = {"eins\n", "zwei\n"};
   char** sp = sa;           // Zeiger auf char*

   cout << sa[0]    << endl;          // eins
   cout << *sa      << endl;          // eins
   cout << sa[1]    << endl;          // zwei
   cout << sa[1][0] << endl;          // z
   cout << *sp      << endl;          // eins
   cout << sp[1]    << endl;          // zwei

   return 0;
}
```



Zeiger

Zeiger und Funktionen

Parameter der Funktion `main()`

`main()` ist eine spezielle Funktion. Jedes C++-Programm startet definitionsgemäß mit `main()`, so dass `main()` in jedem C++-Programm genau einmal vorhanden sein muss.

`main()` kann nicht überladen oder von einer anderen Funktion aufgerufen werden.

Der Rückgabetyp der Funktion `main()` soll `int` sein, die Funktion ist ansonsten aber implementierungsabhängig. Die folgenden beiden Formen sind mindestens gefordert:

```
int main()
int main(int argc, char *argv[])
```

Meist steht auch die folgende Form zur Verfügung:

```
int main(int argc, char *argv[], char *env[])
```

`argc`	ist die Anzahl der Parameter einschließlich des Programmnamens,
`argv[]`	ein String-Array mit den Parametern und
`env[]`	ein String-Array mit den Umgebungsvariablen.

Es gilt `argv[argc] == NULL`.
`argv[0]` enthält den Programmnamen.

Zeiger und Funktionen

Parameter der Funktion `main()`

Beispiel: Programm echo, das seine Kommandozeilen-Argumente ausgibt

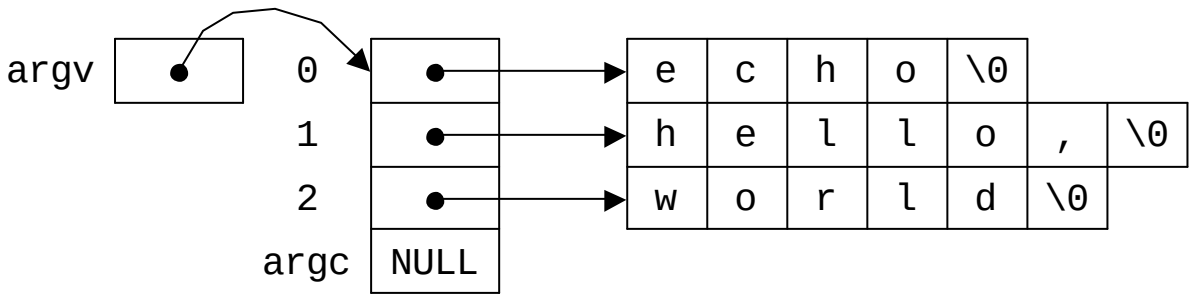
```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    for(int i = 1; i < argc; i++)
        cout << argv[i] << " ";
    cout << endl;
    return 0;
}
```

Der Aufruf

```
echo hello, world
```

gibt aus:

```
hello, world
```



Zeiger und Funktionen

Parameter der Funktion `main()`

Beispiel: Programm echo, das seine Kommandozeilen-Argumente ausgibt

Version 1: Zugriff mit Array-Indizes

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    for(int i = 1; i < argc; i++)
        cout << argv[i] << " ";
    cout << endl;
    return 0;
}
```

Version 2: Zugriff über Zeiger

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    while(--argc > 0)
        cout << *++argv << " ";
    cout << endl;
    return 0;
}
```

Zeiger und Funktionen

Parameter der Funktion `main()`

Beispiel: Kommandozeilen-Argumente und Umgebungsvariablen ausgeben

```
// Programm mainpar.cpp
// Testen z.B. mit dem folgenden Aufruf:
// mainpar first_arg "arg with blanks" 3 4 stop!

#include <iostream>
using namespace std;

int main(int argc, char* argv[], char* env[])
{
    cout << "Aufruf des Programms "
         << argv[0] << endl;

    cout << (argc - 1)
         << " weitere Argumente wurden an"
         << " main() uebergeben:\n";

    int i = 1;
    while(argv[i])
        cout << argv[i++] << endl;

    cout << "\n*** Umgebungsvariablen: ***\n";
    i = 0;
    while(env[i])
        cout << env[i++] << endl;

    return 0;
}
```

Zeiger und Funktionen

Parameter der Funktion `main()`

Der Aufruf

```
mainpar first_arg "arg with blanks" 3 4 stop!
```

könnte z.B. die folgende Ausgabe liefern:

```
Aufruf des Programms = mainpar
5 weitere Argumente wurden an main() uebergeben:
first_arg
arg with blanks
3
4
stop!

*** Umgebungsvariablen: ***
=::=::\
=C:=C:\
=ExitCode=00000000

...
```

Zeiger und Funktionen

Parameterübergabe mit Zeigern

Gewünscht wird eine Funktion `increase()` vom Typ `void`, die den Wert ihres Arguments um 5 erhöhen soll.

```
void increase(int val)      // erwartet einen
{                           // Wert vom Typ int
   val += 5;
}

int main()
{
   int x = 10;              // Der Wert von x
   increase(x);             // wird als Argument
   return 0;                // uebergeben.
}
```

x hat nach der Rückkehr vom Funktionsaufruf immer noch den Wert 10.

Grund: *Wert*übergabe (*call-by-value*)

Zeiger und Funktionen

Parameterübergabe mit Zeigern

Wenn ein übergebenes Objekt modifiziert werden soll, kann die Übergabe *per Zeiger* auf das Objekt geschehen.

```
void increase(int *valp)    // erwartet einen
{                           // Zeiger auf int
   *valp += 5;
}

int main()
{
   int x = 10;              // Die Adresse von x
   increase(&x);            // wird als Argument
   return 0;                // uebergeben.
}
```

Nach der Rückkehr vom Funktionsaufruf hat x den Wert 15.

Dies ist ein Spezialfall der Wertübergabe: innerhalb von `increase()` wird mit einer *Kopie* des Zeigers `&x` gearbeitet, die auf dasselbe Objekt wie das Original zeigt.

Zeiger und Funktionen

Parameterübergabe mit Zeigern

Eine Modifikation des Zeigers in der Funktion `increase()` ändert nicht den Wert des Zeigers in der Funktion `main()`.

```
#include <iostream>
using namespace std;

void increase(int *valp)    // erwartet einen
{                           // Zeiger auf int
   *valp += 5;
   valp++;                  // valp wird modifiziert
   *valp += 1000;
}

int main()
{
   int x[] = {1,2,3,4,5};   // Die Adresse von x[0]
   increase(x);             // wird uebergeben.

   for(int i = 0; i < 5; i++)
      cout << x[i] << "  ";
   cout << endl;

   return 0;
}
```

Ausgabe:

Zeiger und Funktionen

Parameterübergabe mit Referenzen

Wenn ein übergebenes Objekt modifiziert werden soll, kann dies in C++ auch durch Übergabe einer *Referenz* erzielt werden:

```
void increase(int &valr)     // erwartet eine
{                            // Referenz auf int
   valr += 5;
}

int main()
{
   int x = 10;               // Eine Referenz auf x
   increase(x);              // wird als Argument
   return 0;                 // uebergeben.
}
```

Auch hier hat x nach der Rückkehr vom Funktionsaufruf den Wert 15.

Zeiger und Funktionen

Einige Anmerkungen zur Parameterübergabe in C++

Wertübergabe (call-by-value) generell bei nicht zu verändernden Objekten

```
#include <iostream>
using namespace std;

void some_func(int val)
{
   val++;
}

int main()
{
   int x = 10;

   some_func(x);     // x wird nicht veraendert
                     // some_func arbeitet mit
                     // einer Kopie von x
   return 0;
}
```

Ausnahme:

- Große Objekte sollten aus Effizienz- und Speicherplatzgründen nicht per call-by-value übergeben werden.

Zeiger und Funktionen

Einige Anmerkungen zur Parameterübergabe in C++

Falls der Wert eines Arguments geändert werden soll,
Übergabe eines Zeigers oder einer Referenz

```
void by_pointer(int *valp)
{
   *valp += 5;
}

void by_reference(int &valr)
{
   valr += 5;
}

int main ()
{
   int x = 10;

   by_pointer(&x);   // Uebergabe eines Zeigers
   by_reference(x);  // Uebergabe einer Referenz
                     // x kann veraendert werden.
   return 0;
}
```

Zeiger und Funktionen

Einige Anmerkungen zur Parameterübergabe in C++

Falls große Objekte, die nicht verändert werden sollen, in und aus Funktionen übergeben werden, bietet sich die **Übergabe als Referenz auf `const`** an.

```
#include <iostream>
using namespace std;

struct Person
{
   char name[80], address[80];
   double salary;
};

Person person[50];

void printperson(Person const &p)
{
   cout << "Name:    " << p.name << endl;
   cout << "Address: " << p.address << endl;
}

Person const &getperson(int index)
{
   // ...
   return person[index];
}

int main ()
{
   Person boss;

   printperson(boss);
   printperson(getperson(5));
}
```

Zeiger

Zeiger und Funktionen

Gefahren bei der Rückgabe von Zeigern

Wie bei der Rückgabe von Referenzen muss auch bei Zeigern darauf geachtet werden, dass sie nicht auf lokale Objekte verweisen, die nach dem Funktionsaufruf verschwunden sind.

Negativ-Beispiel 1:

```
#include <iostream>
using namespace std;

int *i1()
{
   int x = 123;
   return &x;                // Fehler!
}

int main()
{
   int *ip = i1();
   // Zugriff auf nicht existierendes Objekt
   cout << *ip << endl;
   return 0;
}
```

Zeiger

Zeiger und Funktionen

Gefahren bei der Rückgabe von Zeigern

Negativ-Beispiel 2:

```
#include <iostream>
using namespace std;

char* murks(char* text)
{
   // Speicherplatz reservieren
   char neu[256];
   char *n = neu;
   // text wird nach neu kopiert
   while(*n++ = *text++)
      ;
   return neu;                  // Fehler!
}

int main()
{
   char *strptr = murks("Krach, bumm!");
   // Zugriff auf nicht existierendes Objekt
   cout << strptr << endl;
   return 0;
}
```

Zeiger und Funktionen

Zeiger auf Funktionen

Zeiger auf Funktionen ermöglichen es, erst *zur Laufzeit* zu bestimmen, welche Funktion ausgeführt werden soll (sog. *spätes Binden* oder *dynamisches Binden*).

Beispiel:

```
#include<iostream>
using namespace std;


int max(int x, int y)
{
    return x > y ? x : y;
}

int min(int x, int y)
{
    return x < y ? x : y;
}
```

Zeiger und Funktionen

Zeiger auf Funktionen

```
int main()
{
   int a = 4711, b = 5678;

   // fp ist Zeiger auf eine Funktion,
   // die einen int-Wert zurueckgibt
   // und zwei int-Parameter verlangt
   int (*fp)(int, int);

   do
   {
       char c;
       cout << "max (1) oder min (0) ausgeben "
               "(sonst => Ende) ?";
       cin >> c;

       // Zuweisung von max() oder min()
       // (ohne Angabe von Klammern
       // nach dem Funktionsnamen)
       switch(c)
       {
          case '0' : fp = min; break;
          case '1' : fp = max; break;
          default  : fp = 0;
       }
       if(fp)
       {
          // Dereferenzierung des
          // Funktionszeigers und Aufruf
          cout << (*fp)(a,b) << endl;
          // oder auch direkte Verwendung des Namens
          cout << fp(a,b) << endl;
       }
   } while(fp != 0);
}
```

Zeiger

Dynamische Datenobjekte

C++ bietet die Möglichkeit, mit dem Operator `new` Speicherplatz genau in der richtigen Menge und zum richtigen Zeitpunkt zur Verfügung zu stellen, und diesen Speicherplatz mit `delete` wieder freizugeben, wenn er nicht mehr benötigt wird.

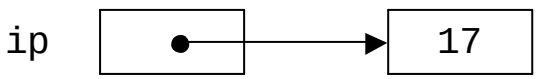
Die mit `new` erzeugten Objekte unterliegen *nicht* den Gültigkeitsbereichsregeln für Variablen.

Der Speicherplatz für mit `new` erzeugte Objekte wird in einem Speicherbereich namens *Halde* (englisch *heap*) reserviert.

Der Zugriff auf die neu auf dem Heap erzeugten Objekte geschieht ausschließlich über Zeiger.

Beispiel:

```
int *ip;
ip = new int;
*ip = 17;
cout << *ip << endl;
```



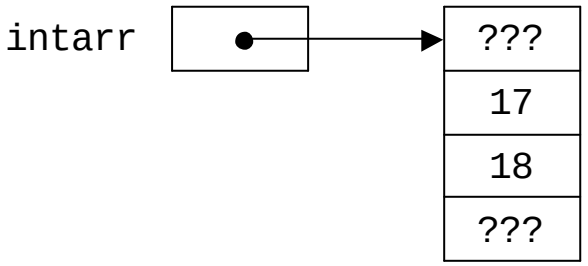
Zeiger

Dynamische Datenobjekte

Mit dem Operator `new[]` kann dynamisch ein Array erzeugt werden. Der Operator `new[]` zur Erzeugung von Arrays wird in C++ vom new-Operator für einzelne Objekte unterschieden.

Beispiel:

```
int *intarr;
intarr = new int [4];
intarr[1] = 17;
intarr[2] = 18;
cout << intarr[2] << endl;
```



Zeiger

Dynamische Datenobjekte

Freigeben dynamischer Objekte

Der Operator `delete` gibt den mit new reservierten Speicherplatz wieder frei. Entsprechend new und `new[]` wird zwischen `delete` und `delete[]` unterschieden.

Beispiele:

```
int *ip;
ip = new int;
*ip = 17;
cout << *ip << endl;
delete ip;

int *intarr;
intarr = new int [4];
intarr[1] = 17;
intarr[2] = 18;
cout << intarr[2] << endl;
delete [] intarr;
```

`delete[]` genau dann verwenden, wenn die Objekte mit `new[]` erzeugt worden sind.

Zeiger

Dynamische Datenobjekte

Freigeben dynamischer Objekte

`delete` darf ausschließlich auf Objekte angewendet werden, die mit `new` erzeugt worden sind.

```
int i;
int *ip = &i;
delete ip;          // Fehler!!!

ip = new int;
delete ip;          // ok
```

`delete` darf nur *einmal* auf ein Objekt angewendet werden.

```
int *ip = new int;
int *ip2 = ip;
delete ip;          // ok

// ip2 ist jetzt ein so genannter haengender
// Zeiger (engl. dangling pointer);
// d.h., ip2 verweist auf einen Speicherbereich
// ohne Bedeutung
cout << *ip;        // Fehler!
cout << *ip2;       // Fehler!
delete ip2;         // Fehler!
```

Zeiger

Dynamische Datenobjekte

Freigeben dynamischer Objekte

`delete` auf einen NULL-Zeiger angewendet bewirkt nichts.

Wenn ein mit `new` erzeugtes Objekt mit dem `delete`-Operator gelöscht wird, wird automatisch der Destruktor für das Objekt aufgerufen.

Mit `new` erzeugte *Objekte* unterliegen *nicht* den Gültigkeitsbereichsregeln für Variable. Sie existieren so lange, bis sie mit `delete` gelöscht werden oder das Programm beendet wird, unabhängig von irgendwelchen Blockgrenzen.

```
{
   int *ip = new int[1000000];
   ip[123456] = 9876;
   // mehr Programmcode,
   // aber kein delete ip;
}
// ip existiert nicht mehr.
// Das Array mit dem ehemaligen Namen *ip
// existiert noch, ist aber nicht mehr erreichbar
// => Speicherleck (englisch memory leak).
```

Zeiger

Dynamische Datenobjekte

Freigeben dynamischer Objekte

korrekt wäre:

```
{
   int *ip = new int[1000000];
   ip[123456] = 9876;
   // mehr Programmcode
   delete [] ip;
}
```

oder:

```
int * ip;     // ausserhalb des Blocks deklariert
{
   ip = new int[1000000];
   ip[123456] = 9876;
   // mehr Programmcode
}
cout << ip[123456];
// mehr Programmcode
delete [] ip;
```

Die Freigabe von Arrays erfordert die Angabe der eckigen Klammern. `delete[]` genau dann verwenden, wenn die Objekte mit `new[]` erzeugt worden sind.

Zeiger

Dynamische Datenobjekte

Die Funktion `set_new_handler()`

Das C++-Laufzeitsystem stellt sicher, dass eine Fehlerbehandlungsroutine aufgerufen wird, wenn die Speicherallokation scheitert.

Als Voreinstellung gibt diese Funktion einen `NULL`-Zeiger an den Aufrufer von `new` zurück.

Die Fehlerbehandlungsroutine kann vom Programmierer redefiniert werden und muss folgende Bedingungen erfüllen:

- Sie hat keine Argumente, und
- sie gibt keinen Wert zurück.
  (Der Rückgabetyp ist `void`.)

Die benutzerdefinierte Fehlerbehandlungsroutine wird im Programm "installiert" durch Aufruf der Funktion `set_new_handler()`.

Dynamische Datenobjekte

Die Funktion `set_new_handler()`

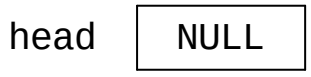
```
#include <cstdlib>          // fuer exit()
#include <new>
#include <iostream>
using namespace std;

void out_of_memory()
{
   cout << "Memory exhausted. "
        << "Program terminates." << endl;
   exit(1);
}

int main()
{
   char *ip;
   long total_allocated = 0;

   // install error function
   set_new_handler(out_of_memory);

   // eat up all memory
   cout << "Ok, allocating..." << endl;
   while(true)
   {
      ip = new char [10000];
      total_allocated += 10000;
      cout << "Now got a total of "
           << total_allocated
           << " bytes." << endl;
   }
   return 0;
 }
```

Dynamische Datenobjekte

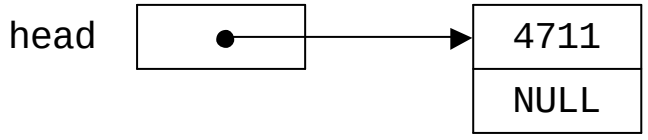
Dynamisch erzeugte Strukturen

Die folgende Struktur Node enthält als zweites Datenelement einen Zeiger next auf ein gleichartiges Objekt. Über diesen Zeiger next können wir eine Verkettung von Node-Objekten erreichen. Dieser Verkettungsmechanismus wird zum Aufbau von Listen verwendet.

```
struct Node {
   int info;
   Node *next;
};

// einen Zeiger auf Node-Objekte anlegen
Node *head = NULL;
```




```
// ein Objekt vom Typ Node erzeugen
// und die Datenelemente initialisieren
// head->next   bedeutet:   (*head).next
head = new Node;
(*head).info = 4711;
head->next = NULL;
```



Dynamische Datenobjekte

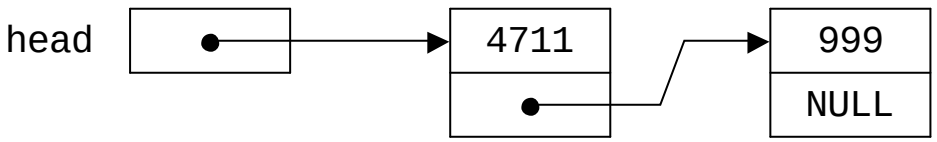
Dynamisch erzeugte Strukturen

Dem Zeiger `head->next` wird mit

```
head->next = new Node;
```

ein weiteres neu erzeugtes Objekt zugewiesen, so dass jetzt zwei miteinander verkettete Objekte vorliegen, die beide über den Zeiger head erreichbar sind.

```
// ein weiteres Objekt vom Typ Node erzeugen
head->next = new Node;
// Zugriff auf Datenelemente des neuen Objekts
head->next->info = 999;
head->next->next = NULL;
```



Dynamische Datenobjekte

Dynamisch erzeugte Strukturen

Beispiel: einfach verkettete Listen

```
#include <iostream>
#include <cstdlib>
using namespace std;


void error(char *s)
{
   cout << s << endl;
   exit(1);
}


struct Node {
   int info;
   Node *next;
};

Node *head = NULL;
```

Dynamische Datenobjekte

Dynamisch erzeugte Strukturen

Beispiel: einfach verkettete Listen

```
void add_front(int info)
{
   Node *tmp;

   tmp = new Node;
   if(tmp == NULL)
      error("add_front: out of memory");
   tmp->info = info;
   tmp->next = head;
   head = tmp;
}

void delete_front()
{
   Node *tmp;

   if(head == NULL)
      error("delete_front: list is empty");
   else {
      tmp = head;
      head = head->next;
      delete tmp;
   }
}
```

Dynamische Datenobjekte

Dynamisch erzeugte Strukturen

Beispiel: einfach verkettete Listen

```
void add_after(Node *p, int info)
{
   if(p == NULL)
      error("add_after: parameter p is NULL");

   Node *tmp = new Node;
   if(tmp == NULL)
      error("add_after: out of memory");
   tmp->info = info;
   tmp->next = p->next;
   p->next = tmp;
}

void delete_after(Node *p)
{
   if(p == NULL)
      error("delete_after: parameter p is NULL");

   else if(p->next == NULL) {
      // der Knoten, auf den p verweist, hat keinen
      // Nachfolger; es gibt also nichts zu tun
   }
   else {
      Node *tmp = p->next;
      p->next = p->next->next;
      delete tmp;
   }
}
```

Dynamische Datenobjekte

Dynamisch erzeugte Strukturen

Beispiel: einfach verkettete Listen

```
int sum_nodes()
{
   Node *pos;
   int sum;

   // Falls die Liste leer ist,
   // wird der Wert Null zurueckgegeben.
   sum = 0;

   pos = head;
   while(pos != NULL)
   {
      sum = sum + pos->info;
      pos = pos->next;
   }
   return sum;
}


// usw. ...
```

Dynamische Datenobjekte

Speicherverwaltung in C vs. C++

In C gibt es statt der Operatoren `new` und `delete` die Funktionen `malloc()` und `free()`.

Unterschiede zwischen `malloc()` und `new`:

- `malloc()` "weiß" nicht, wozu der angeforderte Speicher verwendet wird.
  `new` verlangt die Angabe eines Typs.

- `malloc()` kann den angeforderten Speicherbereich nicht initialisieren.
  `new` ruft beim Anlegen eines neuen Objekts den Konstruktor auf, der evtl. Initialisierungen vornimmt.

- Nach jedem Aufruf von `malloc()` muss überprüft werden, ob `NULL` zurückgegeben wurde.
  `new` stellt einen sogenannten `new_handler` zur Verfügung, der statt expliziter Überprüfungen auf Rückgabe von `NULL` verwendet werden kann.

Unterschiede zwischen `free()` und `delete`:

- `delete` ruft beim Löschen eines Objekts den entsprechenden Destruktor auf.

Dynamische Datenobjekte

Speicherverwaltung in C vs. C++

Beispiel: einfach verkettete Listen in **C**

```
#include <stdio.h>
#include <stdlib.h>


void error(char *s)
{
   printf("%s\n", s);
   exit(1);
}


struct Node {
   int info;
   struct Node *next;
};

struct Node *head = NULL;
```

Dynamische Datenobjekte

Speicherverwaltung in C vs. C++

Beispiel: einfach verkettete Listen in **C**

```
void add_front(int info)
{
   struct Node *tmp;

   tmp = (struct Node*)malloc(sizeof(struct Node));
   if (tmp == NULL)
      error("add_front: out of memory");
   tmp->info = info;
   tmp->next = head;
   head = tmp;
}

void delete_front(void)
{
   struct Node *tmp;

   if (head == NULL)
      error("delete_front: list is empty");
   else {
      tmp = head;
      head = head->next;
      free((void*)tmp);
   }
}
```