Variablen und Konstanten von ANSI C und C# (Teil 3)

Datenstrukturen und Algorithmen entwerfen und anwenden

Inhaltsverzeichnis

1 Einleitung

Wie Sie bis jetzt bereits gelernt haben, stellen ANSI C und auch C# einige einfache (primitive) Datentypen zur Verfügung. So gibt es zum Beispiel für Ganzzahlen die Datentypen: `long`, `int`, `short`, `char` usw. sowie für Fliesskommazahlen die Datentypen: `double`, `float` usw. Es gibt aber auch Fälle, wo diese bestehenden Datentypen nicht mehr ausreichen und wir selber eigene Datentypen zu erstellen haben, um der vorgegebenen Problemstellung gerecht zu werden.

In diesem Dokument wollen wir nun auf die Möglichkeiten eingehen, wie der Programmierer selber eigene Datentypen erstellen kann. Man spricht in diesem Zusammenhang auch von „Benutzerdefinierten Datentypen“.

2 Existierende Datentypen in ANSI C umbenennen (typedef)

Sehr oft ist es hilfreich, einem bestehenden Datentyp einen neuen Namen zu vergeben. So kann eine Umbenennung des Datentyps ein Programm besser lesbar, änderbar und auch besser portierbar machen.

```
typedef  Datentyp  neuerName;   // Angabe zum Datentyp
```

Codeumsetzung 1: Existierende Datentypen in ANSI C umbenennen

2.1 Beispiele zur Datentypenumbenennung (typedef) bei ANSI C

```
// own data types
typedef  unsigned char byte;      // value range: 0 .. 255
typedef  unsigned int  uint;      // positive integer number range
typedef  int  boolean;            // boolean variable with 0 and 1
```

Codebeispiel 1: Datentypenumbenennung in ANSI C

2.2 Hinweise zum Befehl: typedef von ANSI C

- **Der Befehl: typedef existiert in C# nicht mehr!**
- Der Befehl: typedef erzeugt keinen neuen Datentyp, sondern benennt einen bestehen Datentyp um!
- Beim umzubenennenden Datentyp muss es sich um einen in ANSI C gültigen Datentyp handeln, wobei auch selbstdefinierte Datentypen (mittels enum, struct und union erzeugt) möglich sind.
- Die Umbenennung eines bestehenden Datentyps kann auch mittels der #define-Anweisung gemacht werden. So hat zum Beispiel die Anweisung: #define byte unsigned char die gleiche Bedeutung wie die Anweisung: typedef unsigned char byte.

  Trotzdem sollte für die Umbenennung eines bestehenden Datentyps nur die Anweisung: typedef verwendet werden, da diese besser in die Programmiersprache ANSI C integriert ist. Im Weiteren ist typedef flexibler und kann im Gegensatz zu #define gleich bei der Deklaration von Aufzählungstypen, Strukturen und Union neue Datentypen einführen, wie wir noch sehen werden.
- Um einen unbenannten neuen Datentyp überall verwenden zu können, empfiehlt es sich, diesen Typ gleich am Anfang des Programms anzugeben. Per Konvention ist der richtige Ort nach den diversen #include-Anweisungen, aber noch vor main().

3 Aufzählungstyp, Enumeration (`enum`) in ANSI C und C#

Enumerationen sind Aufzählungen von Konstanten, die themenbasiert zusammengehören und alle den gleichen ganzzahligen Datentyp aufweisen. Bei der Deklaration werden ihnen explizit Werte zugewiesen.

Aufzählungstypen sind gedacht für Integer-Variablen, die nicht jeden beliebigen Wert annehmen dürfen, sondern auf eine begrenzte Anzahl von Werten beschränkt sind. Diese einzelnen Werte werden dann über ihren Namen angesprochen.

Ähnlich wie bei den bereits integrierten Konstanten der verwendeten Programmiersprache sind die Namen der Enumerationen und deren Elemente besser lesbar als die durch sie repräsentierten Zahlenwerte. Aus diesem Grund sollten bei der professionellen Programmierung Enumerationen auch öfters angewendet werden.

Innerhalb von Visual Studio gibt es für C# bereits zahlreiche Enumerationen, wie wir später noch feststellen werden. Ein erstes Beispiel dazu stellt die Enumeration: `DialogResult` dar, welche dem Programmierer ermöglicht, die zahlreichen unterschiedlichen Antworten des Benutzers beim Einsatz von Windows-Standarddialogfeldern (Ja, Nein, Abbrechen, Wiederholen, …) anschaulich einzusetzen.

3.1 Generelle Informationen zum Enumerationstyp von ANSI C und C#

Hinweis:

Alle nachfolgenden Code-Beispiele in diesem Abschnitt sind für die Programmiersprache ANSI C angegeben. Selbstverständlich gelten die dazugehörenden Regeln aber auch für die Programmiersprache C#.

- Eine Enumeration sollte keine Leerzeichen enthalten.
- Eine `enum`-Anweisung selbst ist keine Definition einer Variablen, sondern eine Deklaration eines neuen Datentyps. Damit geben Sie dem Compiler bekannt, wie der Aufzählungstyp aufgebaut ist, reservieren aber – wie bei Deklarationen üblich – noch keinen Speicherplatz.
- Jedes Namenssymbol (`Element1`...) steht für einen ganzzahligen Wert, wobei ohne spezielle Angaben durch den Programmierer jedes Namenssymbol um den Wert `1` grösser ist als das Vorhergehende. Der erste Enumerator hat standardmäßig den Wert 0, und der Wert jedes nachfolgenden Enumerators wird jeweils um 1 erhöht. In der folgenden Enumeration gilt z.B.: `Red` ist `0`, `Orange` ist `1` und `Green` ist `2`.

```
enum TrafficLight { Red, Orange, Green };
```

Codebeispiel 2: Deklaration einer Standardenumeration in ANSI C

- Der Anfangswert für das erste Symbol beträgt standardmässig `0`. Dieser Wert kann aber auch abgeändert werden, wenn für das erste Element des Aufzählungstyps ein Wert vorgegeben wird. Wird nur ein Element bestimmt, so erhalten alle nachfolgenden Elemente eine aufsteigende Nummer, die um den Wert `1` grösser ist als sein Vorgänger. In der folgenden Enumeration) gilt z.B.: `Red` ist `10`, `Orange` ist `11` und `Green` ist `12`.

```
enum TrafficLight { Red = 10, Orange, Green };
```

Codebeispiel 3: Deklaration einer Enumeration mit Startwert in ANSI C

- Generell kann jedes Element mit einem Integerwert versehen werden, um so eine eigene Integerzahlzuordnung zu erreichen. Es ist somit möglich, jedem Enumerator einen eigenen Wert mitzugeben. In der folgenden Enumeration gilt: `Red` ist `10`, `Orange` ist `20` und `Green` ist `30`.

```
enum TrafficLight { Red = 10, Orange = 20, Green = 30 };
```

Codebeispiel 4: Deklaration einer Enumeration mit Einzelwertangabe in ANSI C

- Die Elemente einer Aufzählungsliste können überall dort verwendet werden, wo Integer-Konstanten zulässig sind.
- Schreiben Sie alle Elemente (Wertenamen) der Aufzählungsliste zu Anfang mit einem Grossbuchstaben und der Rest des Namens mit Kleinbuchstaben weiter.
- Alle Wertenamen dürfen innerhalb einer Enumeration nur einmal angegeben werden.
- Jede Variable eines Aufzählungstyps darf nur Werte der Aufzählungsliste annehmen.

3.2 Anwendung der Enumeration unter ANSI C

```
enum  [Name] {Element1, Element2, …} [variablenliste]   // Angabe zum Aufz.-Typ
```

Codeumsetzung 2: Erzeugung einer Enumeration mit Variablen in ANSI C

```
typedef enum {Element1, Element2, …}  Name // Erzeugung eines neuen Datentyps
```

Codeumsetzung 3: Erzeugung einer Enumeration als neuen Datentyp in ANSI C

3.2.1 Hinweise zum Erzeugen von Enumerationstypen (`enum`) bei ANSI C

- Um eine Enumeration überall verwenden zu können, empfiehlt es sich, diesen gleich am Anfang des Programms anzugeben. Per Konvention ist der richtige Ort nach den diversen `#include`-Anweisungen, aber noch vor `main()`.
- Wird der Befehl: `enum` innerhalb einer Funktion verwendet, so kann man auch gleichzeitig die Variablen angeben. Bitte beachten Sie aber, dass dieser Aufzählungstyp nur innerhalb dieser Funktion seine Gültigkeit hat!

```
enum Boolean { True, False } answer, solution; // definition of variables
```

Codebeispiel 5: Deklaration und Definition einer Enumeration in ANSI C

- Wird ein Name angegeben, dann kann auf die Variablenliste verzichtet werden, da sich später noch beliebig viele Variablen dieses Aufzählungstyps definieren lassen. Nachfolgendes Beispiel zeigt, wie eine Variable nachträglich definiert werden kann:

```
enum TrafficLight {Red, Orange, Green}; // declaration
enum TrafficLight light1;               // V: light1;  DT: TrafficLight
```

Codebeispiel 6: Deklaration und Definition einer Enumeration in zwei Schritten (ANSI C)

- Um bei der Variablendefinition nicht immer das Schlüsselwort: `enum` verwenden zu müssen, kann mittels dem Schlüsselwort: `typedef` in ANSI C ein eigenständiger Datentyp generiert werden. Diese Verwendungsart von Aufzählungstypen in ANSI C bietet die meiste Flexibilität und sollte wenn möglich immer so verwendet werden. In C# existiert diese Möglichkeit nicht!

```
enum TrafficLight {Red, Orange, Green};      // declaration
typedef enum TrafficLight TrafficStatus;     // data type: TrafficStatus
```

oder kürzer

```
typedef enum {Red, Orange, Green} TrafficStatus; // data type: TrafficStatus
```

Codebeispiel 7: Erzeugung eines eigenen Datentyps einer Enumeration in ANSI C

3.2.2 Beispiele zur Erzeugung von Enumerationstypen bei ANSI C

```
// own data types (enumeration types)
enum TrafficLight {Red, Orange, Green};        // declaration
enum Boolen {True, False} answer, solution;    // definition
enum {On, Off } switch;                        // definition
enum {First = 1, Second, Third} rank;          // definition
```

Codebeispiel 8: Diverse Beispiele von Enumerationen bei ANSI C

3.2.3 Beispiele zum Zugriff auf einen Enumerationswert bei ANSI C

- Das folgende Beispiel zeigt, wie eine Variable: `light1` mit der Ampelfarbe: `Rot` initialisiert wird.

```
TrafficLight light1 = Red;
```

Codebeispiel 9: Wert einer Enumerationsvariablen in ANSI C zuweisen

- Wird ein Enumerationswert ausgegeben, so muss beachtet werden, dass in ANSI C nicht der Wert (z.B. `Rot`) ausgegeben wird sondern der Integerwert, welcher hinter diesem Element steht.

```
printf("%d", light1);  // gives out corresponding integer value
printf("%s", light1);  // creates an error in ANSI C!
```

Codebeispiel 10: Wert einer Enumerationsvariablen in ANSI C ausgeben

Möchte man den Eintrag selbst anzeigen lassen, so muss dies am besten über eine `switch`-Struktur umgesetzt werden, bei der für jeden möglichen Integerwert ein dazugehörender Stringwert vorhanden sein muss.

3.3 Anwendung der Enumeration unter C#

```
enum  Name {Element1, Element2, …}   // Angabe zum Aufzählungs-Typ
```

Codeumsetzung 4: Erzeugung einer Enumeration mit dem Datentyp Integer in C#

```
enum  Name : Datentyp {Element1, Element2, …}   // Angabe zum Aufzählungs-Typ
```

Codeumsetzung 5: Erzeugung einer Enumeration mit beliebigem Ganzzahldatentyp in C#

Um einen Aufzählungstypen überall in einer Klasse verwenden zu können, empfiehlt es sich, diesen Typ gleich am Anfang der Klasse anzugeben.

3.3.1 Hinweise zum Erzeugen von Enumerationstypen (`enum`) bei C#

- In der Regel ist es am besten, eine Enumeration in C# direkt innerhalb eines Namespace zu definieren, damit alle Klassen im Namespace auf die gleiche Weise darauf zugreifen können. Es ist aber auch möglich eine Enumeration innerhalb einer Klasse oder einer Struktur aufzubauen.

- Im Normalfall ist der Datentyp der hinter einer Enumeration steht vom Typ: `int`. Soll dem so sein, so muss kein Datentyp explizit hinter dem Namen angegeben werden.

```
enum Day { Sat, Sun, Mon, Tue, Wed, Thu, Fri };
```

Codebeispiel 11: Deklaration einer Enumeration mit dem Datentyp: Integer

- Sie haben aber die Möglichkeit den Datentyp der Enumeration zu verändern. Die zulässigen Datentypen für eine Enumeration sind: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` oder `ulong`.

  Der zugrunde liegende Typ legt fest, wie viel Speicher für jeden Enumerator reserviert wird.

  Im folgenden Beispiel wird der Datentype auf Byte abgeändert:

```
enum Day : byte { Sat, Sun, Mon, Tue, Wed, Thu, Fri };
```

Codebeispiel 12: Deklaration einer Enumeration mit dem Datentyp: Byte

- Beachten Sie bitte, dass Fliesskommadatentypen nicht erlaubt sind und ein Datentyp nur bei C# vergeben werden kann!
- Um eine Enumvariable zu erzeugen, muss die vorher definierte Enumeration als Datentyp angegeben werden.

```
Day day1;          // creating an enumeration variable
```

Codebeispiel 13: Enumerationsvariable in C# erzeugen

3.3.2 Beispiele zur Erzeugung von Enumerationstypen bei C#

```
// own data types (enumeration types)
enum Day : byte { Sat, Sun, Mon, Tue, Wed, Thu, Fri };      // declaration
enum TrafficLight: byte {Off = 0, On = 10, Defect = 20};    // declaration
enum Switch {On, Off};                                      // declaration
enum Rank {First = 1, Second, Third};                       // declaration
```

Codebeispiel 14: Diverse Beispiele von Enumerationen in C#

3.3.3 Beispiele zum Zugriff auf einen Enumerationswert bei C#

- Das folgende Beispiel zeigt, wie eine Enumerationsvariable: `day1` mit dem Tag: `Mon` initialisiert wird.

```
Day day1 = Day.Mon; // variable: day1 has the value: "Mon"
```

Codebeispiel 15: Enumerationsvariable erzeugen und einen Wert direkt in C# zuweisen

- Um einen `enum`-Typ in einen ganzzahligen Typ zu konvertieren, ist eine explizite Typumwandlung erforderlich. Durch die folgende Anweisung wird der `Mon`-Enumerator beispielsweise einer Variablen vom Typ: `int` zugewiesen. Dabei wird für die Konvertierung von enum in `int` eine Typumwandlung (Cast) verwendet.

```
int enumValue = (int) Day.Mon;  // enumvalue = 2;
```

Codebeispiel 16: Konvertierung einer Enumvariable in einen Integerwert in C#

- Anders als in ANSI C kann ein Enumerationswert als Originalwert direkt ausgegeben werden, wie untenstehende Beispiele zeigen:

```
Day day1 = Day.Mon;          // initialising: day1 = Mon
Console.WriteLine(day1);     // Output: Mon
```

oder

```
Console.WriteLine(Day.Son); // direct way -> Output: Son
```

Codebeispiel 17: Enumerationswert in C# ausgeben

4 Strukturen in ANSI C und C#

Strukturen sind in einer modernen Programmiersprache nicht mehr wegzudenken, da sie die Möglichkeit bieten, zusammenhängende Variablen unter einem Namen zu vereinen. Denken wir z.B. an den Begriff: "Adresse" so wird uns bewusst, dass dieser im Normalfall aus den Elementen: `Vorname`, `Nachname`, `Strasse`, `Postleitzahl` und dem `Ort` zusammengesetzt ist. Diese einzelnen Elemente können nun mit einer Struktur zu einem neuen Datentyp: `Adresse` vereint werden.

Merke: ***!***	***Eine Struktur kann nicht nur verschiedene Elemente gleichen Datentyps zusammenfassen, sondern lässt auch zu, dass verschiedene Datentypen vereint werden. Mit der Deklaration einer Struktur entsteht somit ein neuer Datentyp!***

4.1 Unterschied einer Struktur zu einer Klasse in C#

In C# wird eine Struktur meist mittels einer Klasse realisiert. Es gibt aber auch die Möglichkeit eine Struktur in C# umzusetzen. Nachfolgend sollen kurz die Unterschiede einer Struktur und einer Klasse in C# aufgelistet werden.

Es sind dies:

- Strukturen können auch: Konstanten, Arrays, Properties, geschachtelte Strukturen und Konstruktoren wie auch Methoden (ausser Eigenschaftsmethoden, Property-Methoden) enthalten. Wenn jedoch mehrere solche Member erforderlich sind, sollten Sie sich überlegen, den Typ in eine Klasse umzuwandeln.
- Auf Strukturen kann schneller zugegriffen werden als auf Klassen.
- Eine Struktur ist weniger Speicherintensiv. Wird z.B. ein Array mit 1000 Punkt-Objekten belegt, so wird neben den Koordinatenwerten (x, y) auch zusätzlicher Speicher für die Objektreferenzierung verwendet, damit auf jedes Objekt verwiesen werden kann. In diesem Fall wäre eine Struktur weniger speicherintensiv, da diese Referenzierung entfallen würde.
- Strukturen sind vom Wertetyp und nicht vom Verweistyp, wie bei einer Klasse sonst üblich. Das heisst, dass bei der Kopie einer Strukturvariablen alle Einzelelemente der Struktur kopiert werden. Bei einer Klasse wird hingegen nur die Referenz kopiert.
- Die Elemente einer Struktur müssen öffentlich (`public`) zugänglich sein. In der Klasse werden die Elemente (Attribute) jedoch meist mittels dem Zugriffsmodifizierer: `private` geschützt.
- Strukturen können nicht erben oder vererben. Sie können aber verschachtelt werden (Struktur innerhalb einer weiteren Struktur).
- Die eigene Definition eines (parameterlosen) Standardkonstruktors für eine Struktur verursacht einen Fehler! Die Strukturmember (Variablen der Struktur) können nur mit Hilfe eines parametrisierten Konstruktors für alle Strukturmember oder durch jeweils einzelnen Zugriff auf die Member im Anschluss an die Deklaration der Struktur initialisiert werden.

Merke: ***!***	***Strukturen können keinen Konstruktor ohne Parameterübergabe haben. Sie können aber Konstruktoren mit Parameter haben, die alle Variablen versorgen.***

- Es können Arrays mit dem Datentyp einer Struktur erzeugt werden. Im Fall von Klassen handelt es sich um Arrays mit Verweisen.
- Strukturen können eine Schnittstelle implementieren, aber nicht von einer anderen Struktur erben. Aus diesem Grund können Strukturmember nicht als `protected` deklariert werden.

4.2 Deklaration von Strukturen in ANSI C und C#

Da jede Struktur in ihrem Aufbau völlig einzigartig sein kann, muss der Compiler vor ihrem Gebrauch über seine Zusammensetzung aufgeklärt werden. Dies geschieht mit der Deklaration dieser Struktur. Der Compiler weiss dann, wie viel Speicherplatz er bei einer Variablendefinition für diese Variable zur Verfügung stellen muss.

Die Deklaration einer Struktur wird mit der Verwendung des reservierten Wortes `struct` eingeleitet und hat den folgenden Aufbau:

4.2.1 Deklaration einer Struktur in ANSI C

```
struct [Strukturname] {   // Erzeugung einer Struktur mit einer Variablenliste
  Datentyp variablenname;
  Datentyp variablenname;
  …
} [strukturvariablenliste];
```

Codeumsetzung 6: Deklaration einer Struktur mit Variablenliste in ANSI C

```
typedef struct {           // Erzeugung eines neuen Datentyps
  Datentyp variablenname;
  Datentyp variablenname;
  …
} Strukturname
```

Codeumsetzung 7: Erzeugung eines eignen Strukturdatentyps in ANSI C

4.2.1.1 Codebeispiele der Strukturdeklaration und Variablendefinition in ANSI C

```
struct {              // creating a structure without a structure name
  float xPos;
  float yPos;
} point1, point2;     // list of structure variables
```

Codebeispiel 18: Strukturerstellung ohne Strukturname mit Variablendefinition in ANSI C

Hinweis:

Wird auf den Strukturnamen verzichtet, dann wird eine so genannte "anonyme Deklaration" vorgenommen. Dies bedeutet, dass nur Variablen zur Verfügung stehen, die gleich nach der Strukturdeklaration in der Variablenliste angegeben werden. Möchte man hingegen diesen selbstdefinierten Datentyp ein weiteres Mal verwenden, so kann dies nicht mehr geschehen, da mangels Namen kein Bezug auf diese Struktur gemacht werden kann.

```
struct Point {        // creating a structure with a structure name
  float xPos;
  float yPos;
};
struct Point point1;  // generating a new variable after creating the structure
```

Codebeispiel 19: Strukturerstellung mit Strukturname mit nachträglicher Strukturvariablendefinition in ANSI C

```c
struct Point{          // creating a structure with a structure name
  float xPos;
  float yPos;
} point1, point2;      // list of structure variables

struct Point point3;  // generating a new variable after creating the structure
```

Codebeispiel 20: Strukturerstellung mit Strukturname und Strukturvariablendefinition in ANSI C

```c
// structure declaration with different data types
struct Address {       // creating a structure with a structure name
  char surname[31];
  char name[31];
  char street[41];
  int postcode;
  char place[31];
} student, teacher;    // list of structure variables
```

Codebeispiel 21: Strukturerstellung einer Adresse mit zusätzlicher Strukturvariablendefinitionen in ANSI C

```c
// structure declaration as a new data type
typedef struct {                // creating a structure without a structure name
  char surname[31];
  char name[31];
  char street[41];
  int postcode;
  char place[31];
} AddressType;                  // name of structure data type

AddressType student, teacher;   // list of structure variables
```

Codebeispiel 22: Erstellung des Strukturdatentyps einer Adresse in ANSI C

4.2.2 Deklaration einer Struktur in C#

```
public struct Strukturname {
  public Datentyp variablenname;
  public Datentyp variablenname;
  …
}
```

Codeumsetzung 8: Deklaration einer Struktur in C#

4.2.2.1 Codebeispiel der Strukturdeklaration und Variablendefinition in C#

```c
public struct Point        // structure declaration
{
  public float xPos;       // public member variables
  public float yPos;
}

Point p1 ;                 // structure variable: p1 of structure data type: Point
```

Codebeispiel 23: Strukturerstellung mit Strukturvariablendefinition in C#

4.3 Initialisierung einer Struktur in ANSI C und C#

Sobald eine Strukturvariable mit dem Strukturdatentyp definiert wurde, steht einer Initialisierung nichts mehr im Wege, da ab jetzt der Speicherplatz für die Strukturvariable besteht.

Merke: **!**	***Der Name eines einzelnen Strukturelements besteht aus dem Namen der Struktur (`strukturVariablenName`) und seiner eigenen Membervariablen (`memberVariable`); beide Komponenten werden durch den Punktoperator getrennt.***

4.3.1 Initialisierung einer Struktur in ANSI C

```
struct Strukturname strukturVariablenName = {wert1, wert2, …, wert_n};
```

Codeumsetzung 9: Gleichzeitige Initialisierung einer Strukturvariablen in ANSI C

```
struct Strukturname strukturVariablenName;
strukturVariablenName.memberVariable1 = wert1;
strukturVariablenName.memberVariable2 = wert2;
…
strukturVariablenName.memberVariable_n = wert_n;
```

Codeumsetzung 10: Einzelinitialisierung der Membervariablen einer Strukturvariablen in ANSI C

4.3.1.1 Codebeispiele einer Strukturinitialisierung in ANSI C

```c
// creating a structure with a structure name
struct Address
{
  char surname[31];
  char name[31];
  char street[41];
  int postcode;
  char place[31];
};

// initializing variable: person
struct Address person = { "Meier", "Hans", "Sternmatt 11a", 6789, "Luzern" };
```

Codebeispiel 24: Gleichzeitige Initialisierung einer Strukturvariablen in ANSI C

```c
// initializing variable: person
struct Address person;
strcpy(person.surname, "Meier");
strcpy(person.name, "Hans");
strcpy(person.street, "Sternmatt 11a");
person.postcode = 6789;
strcpy(person.place, "Luzern");
```

Codebeispiel 25: Einzelinitialisierung der Membervariablen einer Strukturvariablen in ANSI C

Hinweis:

- *Wie aus obigem Beispiel gut zu erkennen ist, werden bei allen Variablen die keinen einfachen Datentyp beinhalten die Werte nicht direkt mit dem Zuweisungsoperator zugewiesen, sondern mit einer Bibliotheksfunktion. Da ein String ein "Array of char" darstellt, muss für diesen Zweck die Bibliotheksfunktion: `strcpy()` verwendet werden.*

In ANSI C wird für den Kopiervorgang eines Strings die Funktion: `strcpy()` *eingesetzt. Da wir aber in unserem Unterricht zur Entwicklung unserer Programme Visual Studio verwenden und dort als Programmiersprache C++ einsetzen, muss als Funktion:* `strcpy_s()` *eingesetzt werden.*

- *Handelt es sich hingegen um einen einfachen Datentyp wie z.B. Integer, so kann der Zuweisungsoperator direkt verwendet werden. Dies ist am obigen Beispiel an der folgenden Zeile zu erkennen:*

  Beispiel: `person.postcode = 6789;`

4.3.2 Initialisierung einer Struktur in C#

```
Strukturname strukturVariablenName = new Strukturname (wert1, wert2, … wert_n);
```

Codeumsetzung 11: Gleichzeitige Initialisierung einer Struktur mittels Konstruktor in C#

```
Strukturname strukturvariablenname;
strukturVariablenName.memberVariable1 = wert1;
strukturVariablenName.memberVariable2 = wert2;
…
strukturVariablenName.memberVariable_n = wert_n;
```

Codeumsetzung 12: Einzelinitialisierung der Membervariablen einer Struktur in C#

4.3.2.1 Codebeispiele einer Strukturinitialisierung in C#

```
public struct Point                                     // structure declaration
{
  public int xPos;                                      // public member variables
  public int yPos;

  public Point(int p1, int p2)                          // constructor
  {
    xPos = p1;
    yPos = p2;
  }

  public void printPoint(){                             // method
    Console.WriteLine("Point: " + xPos + " / " + yPos);
  }
}
```

Codebeispiel 26: Strukturerstellung mit dazugehörendem Konstruktor und Ausgabemethode in C#

```
static void Main(string[] args)
{
  Point p1 = new Point(2, 3);       // p1 created und initialized with constructor
  Point p2;                         // p2 created without constructor


  p2.xPos = 4;                      // initialising point: p2
  p2.yPos = 6;

  p1.printPoint();                  // call method (printing localisation)
  p2.printPoint();
}
```

Codebeispiel 27: Initialisierungsarten und Methodenaufruf einer Strukturvariablen in C#

```
public struct Address                // structure declaration
{
  public string surname;             // public member variables
  public string name;
  public string street;
  public int postcode;
  public string place;
}

Address address1;                    // address1 created without constructor
address1.surname = "Meier";          // initialising variable: address1
address1.name = "Hans";
address1.street = "Sternmatt 11a"
address1.postcode = 6789;
address1.place = "Luzern";
```

Codebeispiel 28: Initialisierung der Membervariablen einer Strukturvariablen in C#

Hinweis:

In C# muss für die Initialisierung eines Strings nicht die Methode: `strcpy()` *verwendet werden, wie dies in ANSI C üblich ist. In C# wird der Stringwert direkt mittels dem Zuweisungsoperator der Membervariablen übergeben.*

4.4 Kopieren einer Struktur in ANSI C und C#

```
strukturVariable1 = strukturVariable2;  // Kopieren von Strukturvariablen
```

Codeumsetzung 13: Kopieren von Strukturvariablen in ANSI C und C#

Strukturen sind vom Wertetyp. Das heisst, dass bei der Kopie einer Strukturvariablen alle Einzelelemente (Membervariablen) der Struktur kopiert werden.

4.4.1 Codebeispiele der Kopie einer Strukturvariablen in ANSI C und C#

```
// initializing variable: person
struct Address person;               // create structure variables: person & teacher
struct Address teacher;

strcpy(person.surname, "Meier");
strcpy(person.name, "Hans");
strcpy(person.street, "Sternmatt 11a");
person.postcode = 6789;
strcpy(person.place, "Luzern");

teacher = person ;                  //copy of structure variables: person to teacher
```

Codebeispiel 29: Kopie einer Strukturvariablen in ANSI C

```
public struct Address                // structure declaration
{
  public string surname;             // public member variables
  public string name;
  public string street;
  public int postcode;
  public string place;
}

Address address1;                    // address1 created without constructor
Address address2;                    // address2 created without constructor
```

```
address1.surname = "Meier";          // initialising variable: address1
address1.name = "Hans";
address1.street = "Sternmatt 11a"
address1.postcode = 6789;
address1.place = "Luzern";


address2 = address1;                 // //copy of structure variables: address1 to
address2
```

Codebeispiel 30: Kopie einer Strukturvariablen in C#

5 Zeiger (Pointer) in ANSI C

Bis jetzt wurden Variablen, denen Sie z.B. einen Wert zuweisen, immer über ihren Namen angesprochen. Wird dieser Vorgang genauer untersucht, so stellen wir fest, dass eigentlich nicht dem Namen etwas zugewiesen wird, sondern einem Speicherplatz, der sich hinter diesem Namen verbirgt. Die verfügbaren Plätze im Arbeitsspeicher werden durch Speicheradressen eindeutig gekennzeichnet. Mit Hilfe der Pointer (Zeiger) können Sie in ANSI C direkt auf diese Maschinenadressen zugreifen.

Alle objektorientierten Sprachen (z.B. C++, C#, usw.) verwenden Referenzen, welche auf Objekte zeigen. Solche Referenzen sind im eigentlichen Sinne nichts anderes als Zeiger auf Speicheradressen, wo sich diese Objekte befinden.

In diesem Abschnitt wollen wir auf jene Zeiger eingehen, welche wir in ANSI C bei Funktionsaufrufen zwecks Parameterübergabe antreffen werden.

Der Zusammenhang zur Objektorientierung werden wir in späteren Dokumenten, welche die objektorientierte Sprache genauer betrachten, anschauen.

Merke: ***!***	***Zeiger sind Variablen, die eine Speicheradresse eines Datenobjektes (z.B. Variable) oder einer Funktion beinhalten.***

Während der Programmierer in einer Variablen vom Typ: `int` oder `float` numerische Werte speichert, um damit beispielsweise arithmetische Operationen vorzunehmen, repräsentiert der Wert einer Zeiger-Variablen immer die Adresse einer Speicherzelle. Wie gross eine solche Adresse ist und in welchem Format sie abgelegt wird, hängt vom jeweiligen Betriebssystem ab.

5.1 Hinweise zur Verwendung von Zeigern in ANSI C

- Zeiger sind ein machtvolles und notwendiges aber bei falscher Anwendung auch ein sehr gefährliches Instrument. Falsch angewendete Zeiger führen oft erst später zu einem Fehler oder sogar Programmabsturz. Fehler mit Zeigern sind daher nicht immer einfach zu finden.
- Bei fast allen grösseren Programmen kommen Zeiger zum Einsatz, da sie die Möglichkeit bieten, den Code in kürzerer Schreibweise umzusetzen.
- Werden als Funktionsübergabewerte Adressen (Pointers) anstelle der Werte von Variablen übergeben, so können die Variablen nicht nur gelesen, sondern auch verändert werden. Diese Art der Parameterübergabe wird als "Call by Reference" bezeichnet. (Siehe dazu das Dokument: "Funktionen (Methoden) von ANSI C und C#“)
- Der Name eines Arrays (ohne Angaben des Index) stellt immer eine Adresse zum ersten Element dar. *(Siehe dazu das Dokument: "Variablen und Konstanten von ANSI C und C# (Teil 2)“, Abschnitt: "Arrays")*
- Bei der Zeigeranwendung kommen immer wieder zwei unäre Operationen zum Einsatz. Es sind dies der Adressoperator (*&*) und der Inhaltsoperator (*).

  Hinweis:

  Im Dokument: "Operatoren von ANSI C und C#" finden Sie weitere Informationen zu diesen beiden Operatoren.

5.2 Definition von Zeigern (Pointern) in ANSI C

Pointer müssen, wie andere Variablen auch, vor ihrer Verwendung definiert werden. Dies geschieht in der Form:

```
Datentyp  *zeigername;          // Beschreibung des Pointers
```

Codeumsetzung 14: Definition von Zeigern (Pointern) in ANSI C

Der auffälligste und einzige Unterschied zur Definition einer "gewöhnlichen" Variablen ist der Asterisk (*) vor dem Variablennamen.

5.2.1 Codebeispiel zur Definition eines Zeigers in ANSI C

```
// definition of pointer variables
int  *number;      // pointer for variable: number
char *text;        // pointer for variable: text
```

Codebeispiel 31: Definition eines Zeigers in ANSI C

5.2.2 Hinweise zur Definition eines Zeigers in ANSI C

- Da es sich bei Pointern um Variablen handelt, können diese mit Hilfe verschiedener Operatoren manipuliert werden (in der Praxis fast ausschliesslich durch Addition und Subtraktion). So besteht zum Beispiel die Möglichkeit, den Pointer mittels Additionsrechnungen von einem Element eines Arrays zum nächsten zu führen. Um dies zu tun, muss der Pointer aber wissen, wie gross der Datentyp eines Elementes ist. Aus diesem Grund muss jedem Pointer auch ein Datentyp mitgegeben werden. *(Siehe dazu obige Beispiele.)*
- Möchte man den Zeiger auf das nächste Element in einem Array setzen, so kann dies z.B. mit einem Inkrement (`zeiger++`) auf den Zeiger geschehen. Eine andere Möglichkeit besteht darin, den Zeiger um 1 zu erhöhen. Beide Varianten führen aber schlussendlich zur gleichen Speicheradresse:

  `neue Adresse = alte Adresse + Anzahl Bytes des Datentyps`

  Die zweite Möglichkeit hat aber den Vorteil, dass der Zeiger um eine beliebige Zahl erhöht werden kann und somit jedes Element des Arrays einfach aufrufbar wird.
- Leerzeichen vor und nach dem Stern (*) werden vom Compiler ignoriert.

5.3 Initialisierung von Zeigern in ANSI C

Wie bei anderen Variablen auch ist der Inhalt von Pointern nach ihrer Definition zunächst unbestimmt. Während der Gebrauch von nicht initialisierten, gewöhnlichen Variablen in der Regel lediglich zu unsinnigen Ergebnissen führt, kann dies bei Pointern gefährlich werden. Nicht initialisierte Pointer, auch "wilde Pointer" genannt, zeigen auf irgendeine Stelle im Arbeitsspeicher. Die Manipulation der von solchen Pointern betroffenen Speicherbereiche, deren Inhalt und Besitzer gar nicht bekannt sind, hat im Allgemeinen böse Folgen.

Damit dies nicht passiert, kann man auch Pointer mit einem Anfangswert belegen. Eine Möglichkeit "wilde Pointer" zu vermeiden besteht darin, sie gleich als Null-Pointer zu initialisieren. Die Definition eines "Null-Pointers“ drückt sich darin aus, dass er auf keine Adresse irgendeines Datenobjektes zeigt und die Adresse 0 dafür fest reserviert wird.

```
Datentyp  *zeigername = NULL;        // Beschreibung des Pointers
```

Codeumsetzung 15: Definition und Initialisierung eines Zeigers (Pointern) auf NULL in ANSI C

5.3.1 Codebeispiel zur Initialisierung eines Zeigers auf NULL in ANSI C

```
// definition of pointer variable
int *number = NULL;         // pointer for variable: number
```

Codebeispiel 32: Initialisierung eines Zeigers auf NULL in ANSI C

Hinweis:

Die Konstante: Null ist in der Standardbibliothek: `stdio.h` *definiert. Möchte man diese Konstante verwenden, so muss diese Bibliothek unbedingt im Programm eingebunden werden.*

6 Unionen (union) in ANSI C

Bei einer Union handelt es sich um ein Datenobjekt, dessen Speicherplatz von mehreren Variablen verschiedenen Typs genutzt werden kann. Die Eigenschaft einer Union, mehrere Variablen innerhalb des gleichen Speicherbereichs zu versammeln, eröffnet die Möglichkeit, diesen Speicherabschnitt unter "verschiedenen Gesichtspunkten" zu betrachten. Gemeint ist damit, dass vier hintereinanderliegende Bytes wahlweise z.B. als `float`, `long` oder als Charakter-Array mit 4 Elementen betrachtet werden können.

Unionen finden Anwendung für Typenkonvertierungen, wobei im Hinblick auf Portierbarkeit die Abhängigkeit vom verwendeten System beachtet werden muss.

Unionen eignen sich darüber hinaus zur Erstellung von Tabellen, bei denen jedes Feld Werte verschiedenen Typs aufnehmen kann (wie es z.B. bei Tabellenkalkulationen der Fall ist). Wichtig dabei ist, dass über den zuletzt verwendeten Datentyp Buch geführt wird, weil jeder Wert so gelesen werden muss, wie er abgespeichert wurde. Da Unionen diese Möglichkeit nicht anbieten, empfiehlt sich die Verwendung einer Struktur, in die eine Union eingebettet wird.

Während bei einer Struktur für jedes Struktur-Element Speicherplatz reserviert wird, müssen sich bei der Union alle Elemente den gleichen Speicherplatz teilen. Der Speicherbedarf einer Union richtet sich nicht nach der Gesamtgrösse aller Elemente (wie bei der Struktur zu berechnen ist), sondern nach dem Platzbedarf des grössten Elements.

Hinweise:

- *Da aus zeitlichen Gründen der Datentyp:* `union` *nicht im Modul: M411 behandelt werden kann, wird dieser auch nur kurz mit den Haupteigenschaften besprochen. Um zusätzliche Informationen zu erhalten, sei auf weiterführende Literatur hingewiesen.*
- *Auch in der objektorientierten Sprache C# gibt es eine Art eines solchen Datentyps. Man spricht hier von der sogenannten Basisklasse von der alle anderen Klassen abgeleitet werden und heisst:* `object`*.*

  Da alle Klassen mit Attributen im Normalfall als Datentypen verwendet werden können und jede Klasse von der Klasse: `object` *abgeleitet wird, ist somit ein Datentyp der dem* `union` *entspricht gefunden worden. Das heisst, jedes Objekt einer Klasse kann auch mit dem Datentyp:* `object` *definiert werden.*

Codeumsetzung:

Codebeispiele:

Historie

Vers.	Bemerkungen	Verantwortl.	Datum
0.1	- erstes Zusammenführen der Informationen	W. Odermatt	18.07.2005
1.0	- letzte Layoutanpassungen gemacht	W. Odermatt	13.09.2005
1.1	- Korrekturen durchgeführt	W. Odermatt	12.09.2006
1.2	- Korrekturen durchgeführt	W. Odermatt	22.11.2010
1.3	- Korrekturen durchgeführt	W. Odermatt	29.04.2013
2.0	- Anpassungen an neuen Modulinhalt: M403 umgesetzt	W. Odermatt	13.06.2014
3.0	- Anpassungen und Erweiterungen an neue Programmiersprache C#	W. Odermatt	18.02.2016

Referenzunterlagen

LfNr.	Titel / Autor / File / Verlag / ISBN	Dokument-Typ	Ausgabe
1	„Programmiersprache C“, Implementierung C, Grundkurs / H.U. Steck	Theorieunterlagen	2003
2	„Programmieren in C“ / W. Sommergut / Beck EDV-Berater im dtv / 3-423-50158-8	Fachbuch	1997
3	„C Kompakt Referenz“ / H. Herold / Addison Wesley / 3-8273-1984-6	Fachbuch	2002
4	„C in 21 Tagen“ / P. Aitken, B. L. Jones / Markt + Technik / 3-8272-5727-1	Fachbuch	2000
5	„C Programmieren von Anfang an“ / H. Erlenkötter / rororo / 3-499-60074-9	Fachbuch	2001
6	„Programmieren in C“ / B.W. Kernighan, D.M. Ritchie / Hanser / 3-446-25497-3	Fachbuch	1990
7	„Einstieg in Visual C# 2013“ / Thomas Theis / Galileo Computing / 978-3-8362-2814-5	Fachbuch	2014
8	“Visual C# 2012” / Andreas Kühnel / Rheinwerk Computing / 978-3-8362-1997-6	Fachbuch	2013
9	“Objektorientiertes Programmieren in Visual C#” / Peter Loos / Microsoft Press / 3-86645-406-6	Fachbuch	2006