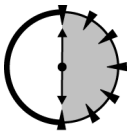
Samstag-nachmittag

Teil 3

Das Array

Checkliste

- ☑ Den Datentyp Array einführen
- ☑ Arrays verwenden
- ☑ Arrays initialisieren
- ☑ Das gebräuchlichste Array verwenden – die Zeichenkette



30 Min.

Die Programme, die wir bisher geschrieben haben, waren immer nur mit einer Zahl beschäftigt. Das Summationsprogramm liest immer eine Zahl von der Tastatur, addiert sie zur bisherigen Summe, die in einer einzigen Variablen gespeichert ist, und liest die nächste Zahl. Wenn wir zu unserer ersten Analogie zurückkehren, dem menschlichen Programm, richten sich diese Programme auf jeweils nur eine Radmutter. Es gibt aber Fälle, in denen wir alle Radmuttern speichern wollen, bevor wir damit anfangen, mit ihnen zu arbeiten.

Diese Sitzung untersucht, wie Werte gespeichert werden können, gerade so, wie ein Mechaniker mehrere Radmuttern gleichzeitig halten oder aufbewahren kann.

11.1 Was ist ein Array?

Lassen Sie uns damit beginnen zu untersuchen, warum und wozu *Arrays* gut sind. Ein Array ist eine Folge von Objekten, normalerweise Zahlen, wobei jedes Objekt über einen Offset angesprochen wird.

Betrachten Sie das folgende Problem. Ich brauche ein Programm, das eine Folge von Zahlen von der Tastatur liest. Ich verwende die mittlerweile Standard gewordenen Regel, dass eine negative Eingabe die Folge abschließt. Nachdem die Zahlen eingegeben sind, und erst dann, soll das Programm die Werte in der Standardausgabe ausgeben.

Ich könnte versuchen, Werte in aufeinanderfolgenden Variablen zu speichern:

```
cin >> nV1;
if (nV1 >= 0)
{
    cin >> nV2;
    if (nV2 >= 0)
    {
     ...
```

Wie Sie sehen, kann dieser Ansatz nicht mehr als ein paar Zahlen handhaben.

Ein Array löst das Problem viel schöner:

```
int nV;
int nValues[128];
for (int i = 0; ; i++)
{
    cin >> nV;
    if (nV < 0)
    {
        break;
    }
    nValues[i] = nV;
}
```

Die zweite Zeile des Schnipsels deklariert ein Array `nValues`. Deklarationen von Arrays beginnen mit dem Typ der Array-Elemente, in diesem Fall `int`, gefolgt von dem Namen des Array. Das letzte Element einer Array-Deklaration ist eine öffnende und eine schließende Klammer, die die maximale Anzahl Elemente enthält, die im Array gespeichert werden können. Im Sourcecode-Schnipsel ist `nValues` als Folge von 128 Integerzahlen deklariert.

Der Schnipsel liest eine Zahl von der Tastatur und speichert sie in einem Element von `nValues`. Auf ein einzelnes Element des Array wird zugegriffen über den Namen des Array, gefolgt von Klammern, die den Index enthalten. Die erste Zahl im Array ist `nValues[0]`, die zweite Zahl ist `nValues[1]` usw.

Im Gebrauch repräsentiert `nValues[i]` ***das*** `i`***-te Element. Die Indexvariable*** `i` ***muss eine Zählvariable sein; d.h.*** `i` ***muss entweder*** `int` ***oder*** `long` ***sein. Wenn*** `nValues` ***ein Array von*** `int` ***ist, dann ist*** `nValues[i]` ***vom Typ*** `int`***.***

Zu weiter Zugriff

Mathematiker zählen in ihren Array von 1 ab. Das erste Element eines mathematischen Arrays ist `x(1)`. Die meisten Programmiersprachen beginnen auch bei 1. C++ beginnt das Zählen bei 0. Das erste Element eines C++-Arrays ist `nValues[0]`.

Es gibt einen guten Grund dafür, dass C++ bei 0 anfängt zu zählen, aber Sie müssen sich bis Sitzung 12 gedulden, in der Sie diesen Grund erfahren werden.

Weil die Indizierung von C++ bei 0 beginnt, ist das letzte Element eines Array mit 128 `int`-Elementen `nArray[127]`.

Unglücklicherweise führt C++ keinen Test durch, ob ein verwendeter Index im Indexbereich des Array liegt. C++ wird Ihnen bereitwillig Zugriff auf `nArray[200]` geben. Sogar `nArray[-15]` ist in C++ erlaubt.

Als Illustration stellen Sie sich bitte vor, die Abstände auf den Landstraßen werden mit Hilfe von äquidistant stehenden Strommasten gemessen. (Im Westen von Texas ist das von der Wirklichkeit nicht weit entfernt.) Lassen Sie uns diese Einheit Mastenlänge nennen. Die Straße zu mir nach Hause beginnt an der Abzweigung von der Hauptstraße und führt auf geradem Wege zu meinem Haus. Die Länge dieser Strecke beträgt exakt 9 Mastenlängen.

Wenn wir die Nummerierung bei den Masten an der Landstraße beginnen, dann ist der Mast, der meinem Haus am nächsten steht, der Mast mit der Nummer 10. Dies sehen Sie in Abbildung 1.1.

Ich kann jede Position entlang der Straße ansprechen, indem ich Masten zähle. Wenn wir Abstände auf der Hauptstraße messen, berechnen wir einen Abstand von 0. Der nächste diskrete Punkt ist eine Mastenlänge entfernt usw., bis wir zu meinem Haus kommen, 9 Mastenlängen entfernt.

Ich kann einen Abstand von 20 Mastenlängen von der Hauptstraße entfernt messen. Natürlich liegt dieser Punkt nicht auf der Straße. (Erinnern Sie sich, dass die Straße an meinem Haus endet.) Tatsächlich weiß ich nicht einmal, was Sie dort vorfinden würden. Sie könnten auf der nächsten Hauptstraße sein, auf freiem Feld, oder im Wohnzimmer meines Nachbarn. Diesen Ort zu untersuchen, ist schlimm genug, aber dort etwas abzulegen, ist noch viel schlimmer. Etwas auf freiem Feld abzulegen, ist eine Sache, aber ins Wohnzimmer meines Nachbarn einzubrechen, könnte Sie in Schwierigkeiten bringen.

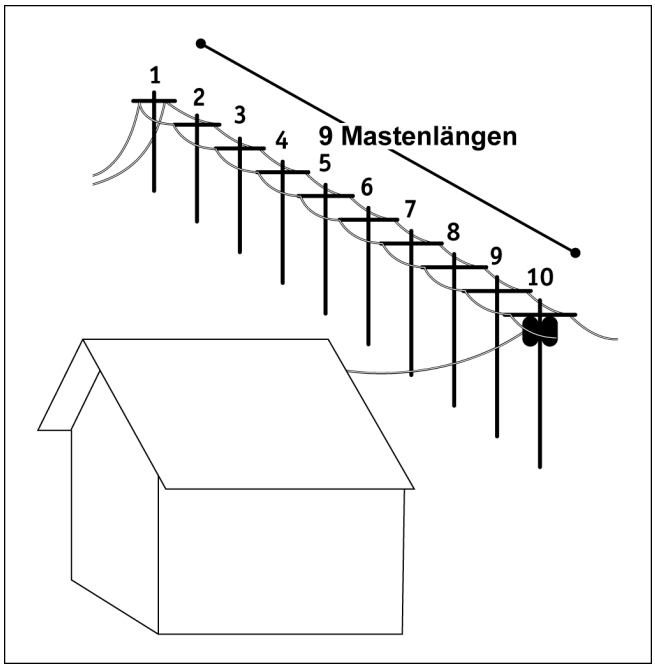
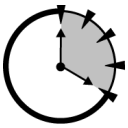


Abbildung 11.1: Man braucht 10 Masten um eine Länge von 9 Mastenlängen abzustecken.

Analog ergibt das Lesen von `array[20]` eines Array mit 10 Elementen einen mehr oder weniger zufälligen Wert. In das Element `array[20]` zu schreiben, hat ein unvorhersehbares Ergebnis. Es kann gut gehen, es kann zu einem fehlerhaften Verhalten führen oder sogar zum Absturz des Programms.

Das Element, auf das in einem Array `nArray` ***mit 128 Elementen am häufigsten illegal zugegriffen wird, ist*** `nArray[128]`. ***Obwohl es nur ein Element außerhalb des Arrays liegt, ist der Zugriff darauf ebenso gefährlich wie das Anfassen jeder anderen nicht korrekten Adresse.***



20 Min.

11.1.2 Ein Array in der Praxis

Das folgende Programm löst das anfänglich gestellte Problem. Das Programm liest eine Folge von Integerwerten von der Tastatur ein, bis der Benutzer eine negative Zahl eingibt. Das Programm zeigt dann alle eingegebenen Zahlen und ihre Summe.

```
// ArrayDemo - demonstriert die Verwendung von Arrays
//             durch Einlesen von Zahlen, die dann
//             in der gleichen Reihenfolge ausgegeben
//             werden
#include <stdio.h>
#include <iostream.h>

// Prototypdeklarationen
int sumArray(int nArray[], int nSize);
void displayArray(int nArray[], int nSize);

int main(int nArg, char* pszArgs[])
{
    // Initialisierung der Summe
    int nAccumulator = 0;
    cout << »Dieses Programm summiert Zahlen,«
         << »die der Benutzer eingibt\n«;
    cout << »Beenden Sie die Schleife durch «
         << »Eingabe einer negativen Zahl\n«;

    // speichere Zahlen in einem Array
    int nInputValues[128];
    int nNumValues = 0;
    do
    {
        // hole nächste Zahl
        int nValue;
        cout << »Nächste Zahl: »;
        cin  >> nValue;

        // wenn sie negativ ist...
        if (nValue < 0)    // Kommentar A
        {
            // ... dann Abbruch
```

```
            break;
        }

        // ... ansonsten speichere die Zahl
        // im Array nInputValues
        nInputValues[nNumValues++] = nValue;
    } while(nNumValues < 128); // Kommentar B

    // Ausgabe der Werte und der Summe
    displayArray(nInputValues, nNumValues);
    cout << »Die Summe ist »
         << sumArray(nInputValues, nNumValues)
         << »\n«;
    return 0;
}
// displayArray - zeige die Elemente eines
//                Array der Länge nSize
void displayArray(int nArray[], int nSize)
{
    cout << »Der Wert des Array ist:\n«;
    for (int i = 0; i < nSize; i++)
    {
        cout.width(3);
        cout << i << »: » << nArray[i] << »\n«;
    }
    cout << »\n«;
}

// sumArray - gibt die Summe der Elemente eines
//            int-Array zurück
int sumArray(int nArray[], int nSize)
{
    int nSum = 0;
    for (int i = 0; i < nSize; i++)
    {
        nSum += nArray[i];
    }
    return nSum;
}
```


Das Programm ArrayDemo beginnt mit der Deklaration eines Prototyps der Funktionen `sumArray( )` und `displayArray( )`. Der Hauptteil des Programms enthält die typischen Eingabeschleifen. Dieses Mal jedoch werden die Werte im Array `nInputValues` gespeichert, wobei die Variable `nNumValues` die Anzahl der bereits im Array gespeicherten Werte enthält. Das Programm liest keine weiteren Werte ein, wenn der Benutzer eine negative Zahl eingibt (Kommentar A), oder wenn die Anzahl der Elemente im Array erschöpft ist (das ist der Test bei Kommentar B).

Das Array** `nInputValues` **ist als 128 Integerzahlen lang deklariert. Sie können denken, dass das genug ist für jedermann, aber verlassen Sie sich nicht darauf. Mehr Werte in ein Array zu schreiben, als es speichern kann, führt zu einem fehlerhaften Verhalten des Programms und oft zum Programmabsturz. Unabhängig davon, wie groß Sie das Array machen, bauen Sie immer einen Check ein, der sicherstellt, dass Sie nicht über die Grenzen des Array hinauslaufen.

Die Funktion `main( )` endet mit der Ausgabe des Array-Inhaltes und der Summe der eingegebenen Zahlen. Die Funktion `displayArray( )` enthält die typische `for`-Schleife, die zum Traversieren eines Array verwendet wird. Beachten Sie, dass der Index mit 0 und nicht mit 1 initialisiert wird. Beachten Sie außerdem, dass die for-Schleife abbricht, bevor `i` gleich `nSize` ist.

In gleicher Weise iteriert die Funktion `sumArray( )` in einer Schleife durch das Array und addiert alle Werte zu der in `nSum` enthaltenen Summe. Nur um Nichtprogrammierer nicht weiter raten zu lassen, der Begriff *»iterieren«* meint das Traversieren durch eine Menge von Objekten wie das Array. Wir sagen »die Funktion `sumArray( )` iteriert durch das Array.«

11.1.3 Initialisierung eines Array

Ein Array kann initialisiert werden, wenn es deklariert wird.

Eine nicht initialisierte Variable enthält einen zufälligen Wert.

Der folgende Codeschnipsel zeigt, wie das gemacht wird:

```
float fArray[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Dies initialisiert `fArray[0]` mit 0, `fArray[1]` mit 1, `fArray[2]` mit 2, usw. Die Anzahl der Initialisierungskonstanten kann auch gleichzeitig die Größe des Array definieren. Z.B. hätten wir durch Zählen der Konstanten feststellen können, dass `fArray` fünf Elemente hat. C++ kann auch zählen. Die folgende Deklaration ist identisch mit der obigen:

```
float fArray[] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Es ist nicht nötig, den gleichen Wert immer wieder zu wiederholen, um ein großes Array zu initialisieren. Das folgende initialisiert die 25 Einträge in `fArray` mit 1.0:

```
float fArray[25] = {1.0};
```

11.1.4 Warum Arrays benutzen?

Oberflächlich gesehen tut das Programm ArrayDemo nicht mehr als unser früheres Programm, das kein Array verwendete. Klar, diese Version kann ihre Eingabe wiederholend ausgeben, bevor die Summe ausgeben wird, ist aber sehr umständlich.

Die Möglichkeit, Eingabewerte wieder anzuzeigen, weist geradezu auf einen entscheidenden Vorteil von Arrays hin. Arrays gestatten einem Programm, eine Reihe von Zahlen mehrfach zu bear-

beiten. Das Hauptprogramm war in der Lage, das Array der Eingabewerte an die Funktion `displayArray( )` zur Darstellung zu übergeben und das gleiche Array an die Funktion `sumArray( )` zur Summenbildung zu übergeben.

11.1.5 Arrays von Arrays

Arrays sind Meister darin, Folgen von Zahlen zu speichern. Einige Anwendungen erfordern Folgen von Folgen. Ein klassisches Beispiel einer Matrixkonfiguration ist die Tabelle. Ausgelegt wie ein Schachbrett erhält jedes Element der Tabelle einen x-Offset und einen y-Offset.

C++ implementiert die Matrix wie folgt:

```
int nMatrix[2][3];
```

Diese Matrix hat zwei Elemente in der einen Dimension und 3 Elemente in der anderen Dimension, also 6 Elemente. Wie Sie erwarten werden, ist eine Ecke der Matrix `nMatrix[0][0]`, während die andere Ecke `nMatrix[1][2]` ist.

Ob Sie `nMatrix` 10 Elemente lang machen in der einen oder der anderen Dimension, ist eine Frage des Geschmacks.

Eine Matrix kann in der gleichen Weise initialisiert werden, wie ein Array:

```
int nMatrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Dies initialisiert das Array `nMatrix[0]`, das drei Elemente besitzt, mit den Werten 1, 2 und 3, und die drei Elemente des Array `nMatrix[1]` mit den Werten 4, 5 und 6.

11.2 Arrays von Zeichen

Die Elemente eines Arrays können von jedem beliebigen C++-Variablentyp sein. Arrays mit Elementen vom Typ `float`, `double` und `long` sind möglich. Arrays von `char`, d.h. Arrays von Zeichen, haben eine besondere Bedeutung.

Ein Array von Zeichen, das meinen Vornamen enthält, würde so aussehen:

```
char sMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'};
```

Das folgende kleine Programm gibt meinen Namen auf einem MS-DOS-Fenster aus, der Standardausgabe.

```
// CharDisplay - Ausgabe eines Zeichenarray über die
//               Standardausgabe, das MS-DOS-Fenster
#include <stdio.h>
#include <iostream.h>

// Prototypdeklarationen
void displayCharArray(char sArray[], int nSize);

int main(int nArg, char* pszArgs[])
{
```

```
    char cMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'};
    displayCharArray(cMyName, 7);
    cout << »\n«;
    return 0;
}

// displayCharArray - gibt ein Zeichenarray
//                    Zeichen für Zeichen aus
void displayCharArray(char sArray[], int nSize)
{
    for(int i = 0; i< nSize; i++)
    {
        cout << sArray[i];
    }
}
```

Das Programm läuft gut, es ist aber unbequem, die Länge eines Array zusammen mit dem Array selber zu übergeben. Wie haben dieses Problem bei der Eingabe von Integerzahlen dadurch vermieden, dass wir die Regel aufgestellt haben, dass eine negative Zahl das Ende der Eingabe bedeuten soll. Wenn wir hier dasselbe machen könnten, müssten wir nicht die Länge des Array mit übergeben – wir würden dann wissen, dass das Array zu Ende ist, wenn dieses besondere Zeichen angetroffen wird.

Lassen Sie uns den Code 0 verwenden, um das Ende eines Zeichenarray zu markieren.

Das Zeichen, dessen Wert 0 ist, ist nicht das gleiche wie 0. Der Wert von 0 ist 0x30. Das Zeichen, dessen Wert 0 ist, wird oft als `\0` ***geschrieben, um den Unterschied klar zu machen. In gleicher Weise ist*** `\y` ***das Zeichen, das den Wert y hat. Das Zeichen*** `\0` ***wird auch Nullzeichen genannt.***

Unter Verwendung dieser Regel sieht unser kleines Programm so aus:

```
// DisplayString - Ausgabe eines Zeichenarrays
//                 über die Standardausgabe, das
//                 MS-DOS-Fenster
#include <stdio.h>
#include <iostream.h>

// Prototypdeklarationen
void displayString(char sArray[]);

int main(int nArg, char* pszArgs[])
{
    char cMyName[] =
        {'S', 't', 'e', 'p', 'h', 'e', 'n', '\0'};
    displayString(cMyName);
    cout << »\n«;
    return 0;
}

// displayString - gibt eine Zeichenkette
//                 Zeichen für Zeichen aus
void displayString(char sArray[])
```

```
{
    for(int i = 0; sArray[i] != 0; i++)
    {
        cout << sArray[i];
    }
}
```

Die Deklaration von cMyName deklariert das Zeichen-Array mit dem Extrazeichen '\0' am Ende. Das Programm DisplayString iteriert durch das Zeichen-Array, bis das Nullzeichen angetroffen wird.

Die Funktion `displayString( )` ist einfacher zu benutzen als ihr Vorgänger `displayCharArray( )`. Es ist nicht mehr nötig, die Länge des Zeichen-Array mit zu übergeben. Weiterhin arbeitet `displayString( )` auch dann, wenn die Länge der Zeichenkette zur Compilezeit nicht bekannt ist. Z.B. wäre das der Fall, wenn der Benutzer eine Zeichenkette über die Tastatur eingeben würde.

Ich habe den Begriff Zeichenkette verwendet, als wäre es ein fundamentaler Typ, wie `int` oder `float`. Als ich den Begriff eingeführt habe, habe ich auch erwähnt, dass die Zeichenkette eine Variation eines bereits existierenden Typs ist. Wie Sie jetzt sehen können, ist eine Zeichenkette ein Nullterminiertes Zeichenarray.

C++ unterstützt eine optionale, etwas bequemere Art der Initialisierung von Zeichenarrays durch eine in Hochkommata eingeschlossene Zeichenkette. Die Zeile

```
char szMyName[] = »Stephen«;
```

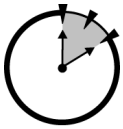
ist exakt äquivalent mit der Zeile

```
char cMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n', '\0'};
```

im vorigen Beispiel.

Die hier verwendete Namenskonvention ist nur eine Konvention, C++ kümmert sich nicht darum. Das Präfix `sz` steht für eine nullterminierte Zeichenkette.

Die Zeichenkette »Stephen« ist acht Zeichen lang, nicht sieben – das Nullzeichen nach dem n wird mitgezählt.



10 Min.

11.3 Manipulation von Zeichenketten

Der C++-Programmierer muss oft Zeichenketten manipulieren.

Obwohl C++ eine Reihe vom Manipulationsfunktionen für Zeichenketten bereitstellt, lassen Sie uns unsere eigenen Funktionen schreiben, um ein Gefühl dafür zu bekommen, wie diese Funktionen arbeiten.

11.3.1 Unsere eigene Verbindungsfunktion

Lassen Sie uns mit einem einfachen, wenn auch etwas längeren C++-Programm beginnen, das zwei Zeichenketten miteinander verbindet.

```
// Concatenate - verbindet zwei Zeichenketten mit
//               einem » - » in der Mitte
#include <stdio.h>
#include <iostream.h>
// die folgende Include-Datei wird für die
// für str-Funktionen benötigt
// #include <string.h>

// Prototypdeklarationen
void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    // lies erste Zeichenkette ...
    char szString1[256];
    cout << »Zeichenkette #1:«;
    cin.getline(szString1, 128);

    // ... nun die zweite Zeichenkette ...
    char szString2[128];
    cout << »Zeichenkette #2:«;
    cin.getline(szString2, 128);

    // ... füge » - » an die erste an ...
    concatString(szString1, » - »);
    // strcat(szString1, » - »);

    // ... füge jetzt die zweite Zeichenkette an ...
    concatString(szString1, szString2);
    // strcat(szString1, szString2);

    // ... und zeige das Ergebnis
    cout << »\n« << szString1 << »\n«;

    return 0;
}

// concatString - hängt die Zeichenkette szSource
//                ans Ende der Zeichenkette szTarget
void concatString(char szTarget[], char szSource[])
{
    // finde das Ende der ersten Zeichenkette
    int nTargetIndex = 0;
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // füge die zweite ans Ende der ersten an
    int nSourceIndex = 0;
```

```
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }

    // füge terminierende Nullzeichen an
    szTarget[nTargetIndex] = '\0';
}
```

Die Funktion `main( )` liest zwei Zeichenketten mittels der Funktion `getline( )`.

Die Alternative `cin >> szString` ***liest bis zum ersten Leerraum. Hier wollen wir bis zum Ende der Zeile lesen.***

Die Funktion `main( )` verbindet die beiden Zeichenketten mittels der Funktion `concatString( )` und gibt dann das Ergebnis aus.

Die Funktion `concatString( )` setzt das zweite Argument, `szSource`, an das Ende des ersten Argumentes, `szTarget`.

Die erste Schleife innerhalb von `concatString( )` iteriert durch die Zeichenkette `szTarget` so lange, bis `nTargetIndex` auf der Null am Ende der Zeichenkette steht.

Die Schleife `while(value != 0)` ***ist das Gleiche wie*** `while(value)`***, weil*** `value` ***als falsch interpretiert wird, wenn es gleich 0 ist, und als wahr wenn es ungleich 0 ist.***

Die zweite Schleife interiert durch die Zeichenkette `szSource` und kopiert die Elemente dieser Zeichenkette in `szTarget`, beginnend mit dem ersten Zeichen von `szSource` und dem Nullzeichen in `szTarget`. Die Schleife endet, wenn `nSourceIndex` auf dem Nullzeichen von `szSource` steht.

Die Funktion `concatString( )` setzt ein abschließendes Nullzeichen ans Ende der Ergebniszeichenkette, bevor sie zurückkehrt.

Vergessen Sie nicht, die Zeichenketten, die Sie in Ihren Programmen selber erzeugen, durch ein Nullzeichen abzuschließen. In der Regel werden Sie dadurch feststellen, dass Sie das vergessen haben, dass die Zeichenkette »Müll« am Ende enthält, oder dadurch, dass das Programm abstürzt, wenn Sie versuchen, die Zeichenkette zu manipulieren.

Das Ergebnis der Programmausführung sieht wie folgt aus:

```
Zeichenkette #1:Die erste Zeichenkette
Enter string #2:DIE ZWEITE ZEICHENKETTE

Die erste Zeichenkette - DIE ZWEITE ZEICHENKETTE
Press any key to continue
```

Es ist sehr verführerisch, C++-Anweisungen wie die folgende zu schreiben:

```
char dash[] = » - »;
concatString(dash, szMyName);
```

Das funktioniert so nicht, weil `dash` ***nur vier Zeichen zur Verfügung hat. Die Funktion wird ohne Zweifel über das Ende des Array*** `dash` ***hinausgehen.***

11.3.2 Funktionen für C++-Zeichenketten

C++ stellt wesentliche Funktionalitäten für Zeichenketten in den Stream-Funktionen >> und << bereit. Sie werden einige dieser Funktionalitäten in Sitzung 28 sehen. Auf einem Basislevel stellt C++ eine Menge einfacher Funktionen bereit, die in Tabelle 11-1 zu sehen sind.

Tabelle 11-1: C++-Bibliotheksfunktionen für die Manipulation von Zeichenketten

Name	Operation
`int strlen(string)`	Gibt die Anzahl der Zeichen einer Zeichenkette zurück
`void strcat(target, source)`	Fügt die `source`-Zeichenkette ans Ende der `target`-Zeichenkette an
`void strcpy(target, source)`	Kopiert eine Zeichenkette in einen Puffer
`int strstr(source1, source2)`	Findet das erste Vorkommen von `source2` in `source1`
`int strcmp(source1, source2)`	Vergleicht zwei Zeichenketten
`int stricmp(source1, source2)`	Vergleicht zwei Zeichenketten, ohne Groß- und Kleinschreibung zu beachten

Im Programm Concatenate hätten wir den Aufruf von `concatString( )` durch einen Aufruf von `strcat( )` ersetzen können, was uns ein wenig Aufwand erspart hätte.

```
strcat(szString, » - »);
```

Sie müssen die Anweisung `#include <string.h>` ***am Anfang Ihres Programms einfügen, das die Funktionen*** `str...` ***verwendet.***

11.3.3 Wide Character

Der C++-Standardtyp `char` ist ein 8-Bit-Feld, das in der Lage ist, Werte von 0 bis 255 darzustellen. Es gibt 10 Ziffern, 26 kleine Buchstaben und 26 große Buchstaben. Selbst wenn die verschiedenen Umlaute und sonstigen Sonderzeichen hinzugefügt werden, ist immer noch genügend Platz, um auch noch das römische und das kyrillische Alphabet unterzubringen.

Probleme mit dem `char`-Typ treten erst dann auf, wenn Sie damit beginnen, asiatische Zeichen, insbesondere japanische und chinesische, darzustellen. Es gibt buchstäblich Tausende dieser Symbole – viel mehr, als sich mit 8 Bit darstellen lassen.

C++ enthält Support für einen neueren Zeichentyp `char` oder `wide character`. Obwohl dies kein elementarer Datentyp wie `char` ist, behandeln ihn mehrere C++-Funktionen, als wäre er das. Z.B. vergleicht `wstrstr( )` zwei Mengen von wchar. Wenn Sie internationale Anwendungen schreiben und auch die asiatischen Sprachen unterstützen wollen, müssen Sie diese `wide-character`-Funktionen verwenden.

11.4 Obsolete Ausgabefunktionen

C++ stellt auch eine Menge von I/O-Funktionen auf einem niedrigen Level bereit. Die nützlichste ist die Ausgabefunktion `printf( )`.

Das sind die originalen I/O-Funkionen von C. Streameingabe und -ausgabe kamen erst mit C++.

In seiner einfachsten Form gibt `printf( )` eine Zeichenkette auf `cout` aus.

```
printf(»Dies ist eine Ausgabe auf cout«);
```

Die Funktion `printf( )` führt Ausgaben unter Verwendung eingebetteter Kommandos zur Formatkontrolle durch, die jeweils mit einem %-Zeichen beginnen. Z.B. gibt das Folgende den Wert einer `int`-Zahl und einer `double`-Zahl aus:

```
int nInt = 1;
double dDouble = 3.5;
printf(»Der int-Wert ist %i; der float-Wert ist %f«,
       nInt, dDouble);
```

Der Integerwert wird an der Stelle von %i eingefügt, der double-Wert erscheint an der Stelle von %f.

```
Der int-Wert ist 1; der float-Wert ist 3.5
```

0 Min.

Obwohl die Funktion `printf( )` kompliziert in ihrer Benutzung ist, stellt Sie doch eine Kontrolle über die Ausgabe dar, die man mit Streamfunktionen nur schwer erreichen kann.

Zusammenfassung

Das Array ist nichts anderes als eine Folge von Variablen. Jede dieser Variablen, die alle den gleichen Typ haben, wird über einen Arrayindex angesprochen – wie z.B. Hausnummern die Häuser in einer Straße bezeichnen. Die Kombination von Arrays und Schleifen, wie `for` und `while`, ermöglichen es Programmen, eine Menge von Elementen einfach zu bearbeiten. Das bekannteste C++Array ist das Null-terminierte Zeichenarray, das auch als Zeichenkette bezeichnet wird.

- Arrays ermöglichen es Programmen, schnell und effizient durch eine Anzahl von Elementen durchzugehen, unter Verwendung eines C++-Schleifenkommandos. Der Inkrementteil einer `for`-Schleife z.B. ist dafür gedacht, einen Index heraufzuzählen, während der Bedingungsteil dafür gedacht ist, auf das Ende des Array zu achten.
- Zugriff auf Elemente außerhalb der Grenzen eines Array ist gleichermaßen häufig anzutreffen wie gefährlich. Es ist verführerisch, auf das Element 128 eines Array mit 128 Elementen zuzugreifen. Weil die Zählung jedoch bei 0 startet, hat das letzte Element den Index 127 und nicht 128.
- Das Abschließen eines Zeichenarrays durch ein spezielles Zeichen erlaubt es Funktionen, zu wissen, wann das Array zu Ende ist, ohne ein spezielles Feld für die Zeichenlänge mitzuführen. Hierfür verwendet C++ das Zeichen '\0', das den Bitwert 0 hat, und kein normales Zeichen ist. Programmierer verwenden den Begriff Zeichenkette oder ASCII-Zeichenkette für eine Null-terminiertes Zeichenarray.
- Der im Abendland entwickelte 8-Bit-Datentyp `char` kann die vielen tausend Spezialzeichen, die in einigen asiatischen Sprachen vorkommen, nicht darstellen. Um diese Zeichen zu verwalten, stellt C++ den speziellen Datentyp `wide character` bereit, der als `wchar` bezeichnet wird. C++ enthält spezielle Funktionen, die mit `wchar` arbeiten können; diese sind in der Standardbibliothek von C++ enthalten.

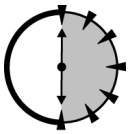
Selbsttest

1. Was ist die Definition eines Array? (Siehe »Was ist ein Array«)
2. Was ist der Offset des ersten und es letzten Elementes im Array `myArray[128]`? (Siehe »Zu weiter Zugriff«)
3. Was ist eine Zeichenkette? Was ist der Typ einer Zeichenkette? Was beendet eine Zeichenkette? (Siehe »Arrays von Zeichen«)

Einführung in Klassen

Checkliste

- ☑ Klassen verwenden, um Variablen verschieden Typs in einem Objekt zu gruppieren
- ☑ Programme schreiben, die Klassen verwenden



30 Min.

Arrays sind großartig, wenn es darum geht, eine Folge von Objekten zu verarbeiten, wie z.B. `int`-Zahlen oder `double`-Zahlen. Arrays arbeiten nicht wirklich gut, wenn Daten verschiedenen Typs gruppiert werden sollen, wie z.B. die Sozialversicherungsnummer und der Name einer Person. C++ stellt hierfür eine Struktur bereit, die als Klasse bezeichnet wird.

12.1 Gruppieren von Daten

Viele der Programme in den vorangegangenen Sitzungen lesen eine Reihe von Zahlen, manche in Arrays, bevor diese verarbeitet werden. Ein einfaches Array ist für allein stehende Zahlen großartig. Es ist jedoch so, dass oft (wenn nicht sogar fast immer) Daten in Form von gruppierten Informationen auftreten. Z.B. könnte ein Programm den Benutzer nach seinem Vornamen, Namen und der Sozialversicherungsnummer fragen – nur in dieser Verbindung machen diese Werte Sinn. Aus einem Grund, der in Kürze klar werden wird, nenne ich eine solche Gruppierung ein *Objekt*.

Eine bestimmte Art, ein Objekt zu beschreiben, nenne ich *parallele Arrays*. In diesem Zugang definiert der Programmierer ein Array von Zeichenketten für den Vornamen, ein zweites für den Nachnamen und ein drittes für die Sozialversicherungsnummer. Diese drei Werte werden über den Index koordiniert.

12.1.1 Ein Beispiel

Das folgende Programm benutzt parallele Arrays, um eine Reihe von Vornamen, Nachnamen und Sozialversicherungsnummern einzugeben und anzuzeigen. `szFirstName[i]`, `szLastName[i]` und `nSocialSecurity[i]` sollen gemeinsam ein Objekt bilden.

```
// ParallelData - speichert zusammengehörende Daten
//                in parallelen Arrays
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// »parallele Arrays « speichern zusammengehörende
// Daten (sind global, damit alle Funktionen Zugriff
// darauf haben)
char szFirstName[25][128];
char szLastName [25][128];
int  nSocialSecurity[25];

// getData - lies einen Namen und eine
//           Sozialversicherungsnummer
//           gib 0 zurück, wenn nichts mehr
//           gelesen werden soll
int getData(int index)
{
    cout << »\nVorname:«;
    cin  >> szFirstName[index];

    // wenn der Vorname 'ende' oder 'ENDE' ist ...
    if ((strcmp(szFirstName[index], »ende«) == 0)
        ||
        (strcmp(szFirstName[index], »ENDE«) == 0))
    {
        // ... gib Kennung für Ende zurück
        return 0;
    }

    // Lade den Rest des Objektes
    cout << »Nachname:«;
    cin  >> szLastName[index];

    cout << »Sozialversicherungsnummer:«;
    cin  >> nSocialSecurity[index];

    return 1;
}

// displayData - gibt den »index«-ten Datensatz aus
void displayData(int index)
{
    cout << szFirstName[index]
         << » »
         << szLastName[index]
         << »/«
         << nSocialSecurity[index]
```

```
        << »\n«;
}

int main(int nArg, char* pszArgs[])
{
    // lade Vornamen, Nachnamen und
    // Sozialversicherungsnummer
    cout << »Lese Vornamen, Nachnamen und\n«
         << »Sozialversicherungsnummer\n«;
         << »Geben Sie 'ende' als Vorname ein, \n«;
         << »um das Programm zu beenden\n«;
    int index = 0;
    while (getData(index))
    {
        index++;
    }

    cout << »\nEinträge:\n«;
    for (int i = 0; i < index; i++)
    {
        displayData(i);
    }
    return 0;
}
```

Die drei koordinierten Arrays sind wie folgt deklariert.

```
char szFirstName[25][128];
char szLastName[25][128];
int sSocialSecurity[25];
```

Die drei Arrays bieten Platz für jeweils 25 Einträge. Vorname und Nachname sind auf 128 Zeichen begrenzt.

Es werden keine Überprüfungen gemacht, die sicherstellen, dass das 128-Zeichen-Limit nicht überschritten wird. In realen Applikationen ist es nicht akzeptabel, auf solche Überprüfungen zu verzichten.

Die Funktion `main( )` liest erst die Objekte in der Schleife ein, die mit `while(getData(index))` in der Funktion `main( )` beginnt. Der Aufruf von `getData( )` liest den nächsten Eintrag. Die Schleife wird verlassen, wenn `getData( )` eine 0 zurückgibt, die anzeigt, dass der Eintrag vollständig ist.

Das Programm ruft dann `displayData( )` auf, um die eingegebenen Objekte auszugeben.

Die Funktion `getData( )` liest Daten von `cin` in die drei Arrays. Die Funktion gibt 0 zurück, wenn der Benutzer einen Vornamen `ende` oder `ENDE` eingibt. Wenn der Vorname davon verschieden ist, werden die verbleibenden Daten gelesen und es wird eine 1 zurückgegeben, um anzuzeigen, dass noch weitere Objekte gelesen werden sollen.

Die folgende Ausgabe stammt von einem Beispiellauf von ParallelData.

```
Lese Vornamen, Nachnamen und
Sozialversicherungsnummer
Geben Sie 'ende' als Vorname ein,
um das Programm zu beenden

Vorname:Stephen
Nachname:Davis
Sozialversicherungsnummer:1234

Vorname:Scooter
Nachname:Dog
Sozialversicherungsnummer:3456

Vorname:Valentine
Nachname:Puppy
Sozialversicherungsnummer:5678

Vorname:ende

Einträge:
Stephen Davis/1234
Scooter Dog/3456
Valentine Puppy/5678
```

12.1.2 Das Problem

Der Zugang der parallelen Arrays ist eine Lösung für das Problem, Daten zu gruppieren. In vielen älteren Programmiersprachen gab es keine Alternative dazu. Für große Datenmengen wird das Synchronisieren möglicherweise vieler Arrays zu einem Problem.

Das einfache Programm ParallelData muss sich nur um drei Arrays kümmern. Denken Sie an die Datenmenge, die eine Kreditkarte pro Datensatz beanspruchen würde. Es würden sicherlich Dutzende von Arrays benötigt.

Ein zweites Problem ist, dass es für einen pflegenden Programmierer nicht offensichtlich ist, dass die Arrays zusammengehören. Wenn der Programmierer nur auf einigen Arrays ein Update durchführt, werden die Daten korrupt.

12.2 Die Klasse

Was benötigt wird, ist eine Struktur, die alle Daten speichern kann, die benötigt werden, um ein einzelnes Objekt zu beschreiben. Ein einzelnes Objekt würde dann den Vornamen, den Nachnamen und die Sozialversicherungsnummer enthalten. C++ verwendet eine solche Struktur, die als Klasse bezeichnet wird.

12.2.1 Das Format einer Klasse

Eine Klasse zur Speicherung eines Vornamens, Namens und einer Sozialversicherungsnummer könnte so aussehen:

```
// die Datensatzklasse
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
        int  nSocialSecurity;
};
// eine einzelne Instanz
NameDataSet nds;
```

Eine Klassendefinition beginnt mit dem Schlüsselwort `class`, gefolgt von dem Namen der Klasse und einem öffnenden/schließenden Klammernpaar.

Das alternative Schlüsselwort `struct` ***kann auch verwendet werden. Die beiden Schlüsselworte*** `class` ***und*** `struct` ***sind identisch, mit der Ausnahme, dass bei*** `struct` ***eine*** `public`***-Deklaration angenommen wird.***

Die erste Zeile innerhalb der Klammern ist das Schlüsselwort `public`.

Spätere Sitzungen werden die anderen Schlüsselworte von C++ außer `public` ***vorstellen.***

Nach dem Schlüsselwort `public` kommen die Einträge, die für die Beschreibung eines Objektes benötigt werden. Die Klasse `NameDataSet` enthält den Vor- und Nachnamen zusammen mit der Sozialversicherungsnummer. Erinnern Sie sich daran, dass eine Klassendeklaration alle Daten umfasst, die ein Objekt beschreiben.

Die letzte Zeile deklariert die Variable `nds` als einen einzelnen Eintrag der Klasse `NameDataSet`. Programmierer sagen, dass `nds` eine Instanz der Klasse `NameDataSet` ist. Sie instanziieren die Klasse `NameDataSet`, um das Objekt `nds` zu erzeugen. Schließlich sagen die Programmierer, dass `szFirstName` und die anderen Elemente oder Eigenschaften der Klasse sind.

Die folgende Syntax wird für den Zugriff auf die Elemente eines Objektes verwendet:

```
NameDataSet nds;
nds.nSocialSecurity = 10;
cin >> nds.szFirstName;
```

Hierbei ist `nds` eine Instanz der Klasse `NameDataSet` (d.h. ein bestimmtes NameDataSet-Objekt). Die Integerzahl `nds.nSocialSecurity` ist eine Eigenschaft des Objektes `nds`. Der Typ von `nds.nSocialSecurity` ist `int`, während der Typ von `nds.szFirstName` gleich `char[]` ist.

Ein Klassenobjekt kann bei seiner Erzeugung wie folgt initialisiert werden:

```
NameDataSet nds = {»Vorname«, »Nachname«, 1234};
```

Der Programmierer kann auch Arrays von Objekten deklarieren und initialisieren:

```
NameDataSet ndsArray[2] = {{»VN1«, »NN1«, 1234}
                           {»VN2«, »NN2«, 5678}};
```

Nur der Zuweisungsoperator ist für Klassenobjekte defaultmäßig definiert. Der Zuweisungsoperator weist dem Zielobjekt eine binäre Kopie des Quellobjektes zu. Beide Objekte müssen denselben Typ besitzen.

12.2.2 Beispielprogramm

Die Klassen-basierte Version des Programms ParallelData sieht wie folgt aus:

```
// ClassData - speichert zusammengehörende Daten
//             in einem Array von Objekten
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet - speichert Vorname, Name und
//               Sozialversicherungsnummer
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
        int  nSocialSecurity;
};
// getData - lies einen Namen und eine
//           Sozialversicherungsnummer
//           gib 0 zurück, wenn nichts mehr
//           gelesen werden soll
int getData(NameDataSet& nds)
{
    cout << »\nVorname:«;
    cin  >> szFirstName[index];

    // wenn der Vorname ‘ende’ oder ‘ENDE’ ist ...
    if ((strcmp(nds.szFirstName, »ende«) == 0)
        ||
        (strcmp(nds.szFirstName, »ENDE«) == 0))
    {
        return 0;
    }

    cout << »Nachname:«;
    cin  >> nds.szLastName;

    cout << »Sozialversicherungsnummer:«;
    cin  >> nds.nSocialSecurity;
```

```
    return 1;
}

// displayData - gib »index«-ten Datensatz aus
void displayData(NameDataSet& nds)
{
    cout << nds.szFirstName
         << » »
         << nds.szLastName
         << »/«
         << nds.nSocialSecurity
         << »\n«;
}

int main(int nArg, char* pszArgs[])
{
    // alloziere 25 Datensätze
    NameDataSet nds[25];

    // lade Vornamen, Nachnamen und
    // Sozialversicherungsnummer
    cout << »Lies Vornamen, Nachnamen und \n«
         << »Sozialversicherungsnummer\n«;
         << »Geben Sie 'ende' als Vorname ein, \n«;
         << »um das Programm zu beenden\n«;
    int index = 0;
    while (getData(nds[index]))
    {
        index++;
    }

    cout << »\nEinträge:\n«;
    for (int i = 0; i < index; i++)
    {
        displayData(nds[i]);
    }
    return 0;
}
```


In diesem Fall alloziert die Funktion `main( )` 25 Objekte aus der Klasse `NameDataSet`. Wie zuvor betritt `main( )` eine Schleife, in der Einträge von der Tastatur gelesen werden unter der Verwendung der Methode `getData( )`. Statt einen einfachen Index zu übergeben (oder einen Index und ein Array), übergibt `main( )` das Objekt, das `getData(NameDataSet)` mit Werten füllen soll.

In gleicher Weise verwendet `main( )` die Funktion `displayData(NameDataSet)`, um alle `NameDataSet`-Objekte anzuzeigen.

Die Funktion `getData( )` liest die Objektinformation in das übergebene Objekt vom Typ `NameDataSet`, das `nds` heißt.

Die Bedeutung des Und-Zeichens (&), das dem Argument der Funktion `getData( )` ***hinzugefügt wurde, wird in Sitzung 13 vollständig erklärt. An dieser Stelle reicht es aus zu erwähnen, dass dieses Zeichen dafür sorgt, dass Änderungen, die in*** `getData( )` ***ausgeführt werden, auch in*** `main( )` ***sichtbar sind.***

0 Min.

12.2.3 Vorteile

Die grundlegende Struktur des Programms ClassData ist die gleiche wie die von ParallelData. ClassData muss jedoch nicht mehrere Arrays verwalten. Bei einem Objekt, das so einfach ist wie `NameDataSet`, ist es nicht offensichtlich, dass dies ein entscheidender Vorteil ist. Überlegen Sie sich einmal, wie beide Programme aussehen würden, wenn `NameDataSet` alle Einträge enthalten würde, die für eine Kreditkarte benötigt würden. Je größer die Objekte, desto größer der Vorteil.

Je weiter wir die Klasse `NameDataSet` ***entwickeln, desto größer wird der Vorteil.***

Zusammenfassung

Arrays können nur Folgen von Objekten gleichen Typs speichern; z.B. ein Array für `int` oder `double`. Die Klasse ermöglicht es dem Programmierer, Daten mit verschiedenen Typen in einem Objekt zu gruppieren. Z.B. könnte eine Klasse `Student` eine Zeichenkette für den Namen des Studenten, eine Integervariable für seine Immatrikulationsnummer und eine Gleitkommazahl für seinen Notendurchschnitt enthalten. Die Kombination von Array und Klassenobjekten kombiniert die Vorteile eines jeden in einer einzelnen Datenstruktur.

- Die Elemente eines Klassenobjektes müssen nicht vom selben Typ sein; wenn Sie verschieden sind, müssen sie über ihren Namen und nicht über einen Index angesprochen werden.
- Das Schlüsselwort `struct` kann an Stelle des Schlüsselwortes `class` verwendet werden; `struct` ist ein Überbleibsel aus den Tagen von C.

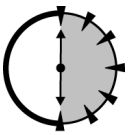
Selbsttest

1. Was ist ein Objekt? (Siehe »Gruppieren von Daten«)
2. Was ist der ältere Begriff für das C++-Schlüsselwort `class`? (Siehe »Das Format einer Klasse«)
3. Was bedeuten die kursiv geschriebenen Worte? (Siehe »Das Format einer Klasse«)
   a. *Instanz* einer Klasse
   b. *Instanziieren* einer Klasse
   c. *Element* einer Klasse

Einstieg C++-Zeiger

Checkliste

- ☑ Variablen im Speicher adressieren
- ☑ Den Datentyp Zeiger einführen
- ☑ Die inhärente Gefahr von Zeigern erkennen
- ☑ Zeiger an Funktionen übergeben
- ☑ Objekte frei allozieren, bekannt als Heap



30 Min.

Teil II führte die C++-Operatoren ein. Operationen wie Addition, Multiplikation, bitweises AND und logisches OR wurden auf elementaren Datentypen wie `int` und `float` ausgeführt. Es gibt einen anderen Variablentyp, den wir noch betrachten müssen – Zeiger.

Für jemanden, der mit anderen Programmiersprachen vertraut ist, sieht C++ bis jetzt aus wie jede andere Programmiersprache. Viele Programmiersprachen enthalten nicht die dargestellten logischen Operatoren und C++ bringt seine eigene Semantik mit, aber es gibt keine Konzepte in C++, die nicht in anderen Sprachen auch vorhanden wären. Bei der Einführung von Zeigern in die Sprache verabschiedet sich C++ von anderen, konventionelleren Sprachen.

Zeiger wurden tatsächlich im Vorgänger von C++, der Sprache C eingeführt. Alles in diesem Kapitel geht in C genauso.

Zeiger sind nicht nur ein Segen. Während Zeiger C++ einmalige Fähigkeiten verleihen, können sie syntaktisch kompliziert sein und sind eine häufige Fehlerquelle. Dieses Kapitel führt den Variablentyp Zeiger ein. Es beginnt mit einigen konzeptionellen Definitionen, geht durch die Syntax von Zeigern, und stellt dann einige Probleme dar, die Programmierer mit Zeigern haben können.

13.1 Was ist deine Adresse?

Wie in der Aussage »jeder muss irgendwo sein«, ist jede Variable irgendwo im Speicher des Computers vorhanden. Der Speicher ist aufgeteilt in einzelne Bytes, wobei jedes Byte seine eigene Adresse hat, 0, 1, 2, 3 usw.

Es gilt die Konvention, Speicheradressen hexadezimal zu schreiben.

Verschiedene Variablentypen benötigen unterschiedlich viele Bytes im Speicher. Tabelle 13-1 zeigt den Speicher, der von den Variablentypen in Visual C++ 6 und GNU C++ auf einem Pentium-Prozessor benötigt wird.

Tabelle 13-1: Speicherbedarf verschiedener Variablentypen

Variablentyp	Speicherbedarf [Bytes]
`int`	4
`long`	4
`float`	4
`double`	8

Betrachten Sie das folgende Testprogramm Layout, das die Anordnung der Variablen im Speicher illustriert. (Ignorieren Sie den Operator & – es ist ausreichend zu sagen, dass `&n` die Adresse der Variablen `n` zurückgibt.)

```
// Layout - dieses Programm versucht einen Eindruck
//          davon zu vermitteln, wie Variablen im
//          Speicher angeordnet sind
#include <stdio.h>
#include <iostream.h>

int main(int nArgc, char* pszArgs[])
{
    int    m1;
    int    n;
    long   l;
    float  f;
    double d;
    int    m2;

    // setze Ausgabemodus auf hexadezimal
    cout.setf(ios::hex);
```

```
// gib die Adresse jeder Variable aus
    // in der obigen Reihenfolge, um einen
    // Eindruck von ihrer Größe zu bekommen
    cout << »-- = 0x« << (long)&m1 << »\n«;
    cout << »&n  = 0x« << (long)&n  << »\n«;
    cout << »&l  = 0x« << (long)&l  << »\n«;
    cout << »&f  = 0x« << (long)&f  << »\n«;
    cout << »&d  = 0x« << (long)&d  << »\n«;
    cout << »-- = 0x« << (long)&m2 << »\n«;

    return 0;
}
```

Die Ausgabe dieses Programms finden Sie in Listing 13-1. Sie können sehen, dass die Variable `n` an der Adresse 0x65fdf0 gespeichert wurde.

Machen Sie sich keine Sorgen, wenn die Werte, die von Ihrem Programm ausgegeben werden, davon verschieden sind. Nur der Abstand zwischen den Adressen ist entscheidend.

Aus dem Vergleich der Adressen können wir ableiten, dass die Größe von `n` gleich 4 Bytes ist (0x65fdf4 – 0x65fdf0), die Größe der `long`-Variablen `l` ebenfalls gleich 4 ist (0x65fdf0 – 0x65fdec) usw.

Das ist aber nur dann so richtig, wenn wir annehmen können, dass die Variablen unmittelbar hintereinander im Speicher angeordnet werden, was bei GNU C++ der Fall ist. Für Visual C++ ist dafür eine besondere Projekteinstellung nötig.

Listing 13-1: Ausgaben des Programms Layout

```
--    = 0x65fdf4
&n    = 0x65fdf0
&l    = 0x65fdec
&f    = 0x65fde8
&d    = 0x65fde0
--    = 0x65fddc
```

Es gibt nichts in der Definition von C++, das vorschreiben würde, dass die Variablen wie in Listing 13-1 angeordnet sein müssen. Was uns betrifft ist es ein großer Zufall, dass GNU C++ und Visual C++ die gleiche Anordnung der Variablen im Speicher gewählt haben.

13.2 Einführung in Zeigervariablen

Lassen Sie uns mit einer neuen Definition und einer Reihe neuer Operatoren beginnen. Eine Zeigervariable ist eine Variable, die eine Adresse enthält, im Allgemeinen die Adresse einer anderen Variable. Zu dieser Definition gehören die neuen Operatoren in Tabelle 13-2.

Tabelle 13-2: Operatoren für Zeiger

Operator	Bedeutung
& (unär)	die Adresse von
* (unär)	(in einem Ausdruck) das, worauf gezeigt wird (in einer Deklaration) Zeiger auf

Die Verwendung dieser Features wird am besten in einem Beispiel deutlich:

```
void fn()
{
   int nInt;
   int* pnInt;

   pnInt  = &nInt;    // pnInt zeigt nun auf nInt
   *pnInt = 10;       // speichert 10 in int-Platz,
                     // auf den pnInt zeigt
}
```

Die Funktion `fn( )` beginnt mit der Deklaration von `nInt`. Die nächste Anweisung deklariert eine Variable `pnInt`, die vom Typ »Zeiger auf `int`« ist.

Zeigervariablen werden wie normale Variablen deklariert, mit Ausnahme des Zeichens *. Dieses Zeichen kann irgendwo zwischen dem Namen des Basistyps, in diesem Falle `int`, und dem Variablennamen stehen. Es wird jedoch zunehmend üblich, den Stern an das Ende des Variablentyps zu setzen.

Wie die Namen von `int`-Variablen mit n beginnen, ist der erste Buchstabe von Zeigervariablen p. Somit ist `pnX` ein Zeiger auf eine `int`-Variable `X`. Auch das ist nur eine Konvention, um den Überblick zu behalten – C++ kümmert sich nicht um die Namen, die Sie verwenden.

In einem Ausdruck bedeutet der unäre Operator & »Adresse von«. Somit würden wir die erste Zuweisung lesen als »speichere die Adresse von `nInt` in `pnInt`«.

Um es noch konkreter zu machen, nehmen wir an, dass der Speicherbereich von `fn( )` bei Adresse 0x100 beginnt. Lassen Sie uns weiterhin annehmen, dass nInt an Adresse 0x102 und pnInt an Adresse 0x106 steht. Die Anordnung hier ist einfacher als die Ausgabe des Programms Layout, aber die Konzepte sind identisch.

Die erste Zuweisung wird in Abbildung 13.1 dargestellt. Hier können Sie sehen, dass der Wert von `&nInt` (0x102) in `pnInt` gespeichert wird.

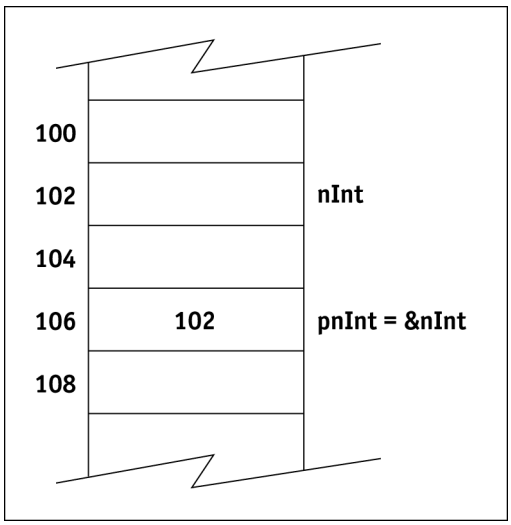


Abbildung 13.1: Speichern der Adresse von nInt in pnInt.

Die zweite Zuweisung in dem kleinen Programmschnipsel besagt, »speichere 10 in der Adresse, auf die `pnInt` zeigt«. Abbildung 13.2 demonstriert dies. Der Wert 10 wird an der Adresse gespeichert, die `pnInt` enthält; diese ist 0x102 (die Adresse von `nInt`).

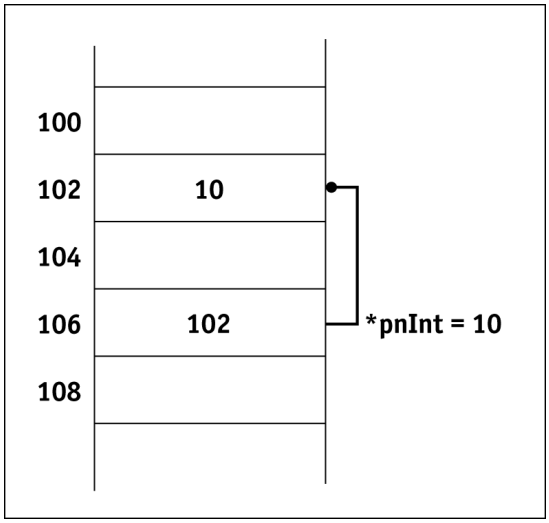
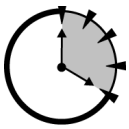


Abbildung 13.2: Speichern von 10 in der Adresse, auf die pnInt zeigt.



20 Min.

13.3 Typen von Zeigern

Erinnern Sie sich daran, das jeder Ausruck einen Typ und einen Wert hat. Der Typ des Ausdrucks `&nInt` ist ein Zeiger auf Integer, der als `int*` geschrieben wird. Ein Vergleich dieses Ausdruckstyps mit der Deklaration von `pnInt` zeigt, dass die Typen exakt zueinander passen.

```
pnInt = &nInt; // beide Seiten sind vom Typ int*
```

In gleicher Weise ist der Typ von `*pnInt` gleich `int`, weil `pnInt` von Typ `int*` ist.

```
*pnInt = 10; // beide Seiten sind vom Typ int
```

Sprachlich ausgedrückt ist der Typ von etwas, auf das `pnInt` verweist, gleich `int`.

Abgesehen davon, dass eine Zeigervariable einen unterschiedlichen Typ hat, wie `int*` und `double*`, ist der Zeiger an sich ein elementarer Typ. Unabhängig, auf was er zeigt, benötigt ein Zeiger auf einem Rechner mit Pentiumprozessor 4 Bytes.

Es ist extrem wichtig, dass Typen zusammenpassen. Bedenken Sie, was passieren könnte, wenn das Folgende erlaubt wäre:

```
int n1;
int* pnInt;
pnInt = &n1;
*pnInt = 100.0;
```

Die zweite Zuweisung versucht, einen 8-Bit-Wert 100.0 von Typ `double` in den 4 Bytes zu speichern, die von der Variable `n1` belegt werden. Das Ergebnis ist, dass Variablen in der Nähe ausgelöscht werden. Das wird grafisch im folgenden Programm LayoutError dargestellt, das Sie in Listing 13-2 finden. Die Ausgabe dieses Programms finden Sie am Ende des Listings.

```
// LayoutError - demonstriert das Ergebnis, wenn
//               Zeiger falsch verwendet werden
#include <stdio.h>
#include <iostream.h>

int main(int nArgc, char* pszArgs[])
{
    int    upper = 0;
    int    n     = 0;
    int    lower = 0;

    // gib die Werte der Variablen vorher aus ...
    cout << »darüber  = » << upper << »\n«;
    cout << »n        = » << n     << »\n«;
    cout << »darunter = » << lower << »\n«;

    // nun speichere einen double-Wert im Speicher,
    // der für ein int alloziert wurde
    cout << »\ndouble wird zugewiesen\n«;
    double* pD = (double*)&n;
```

```
    *pD = 13.0;

    // Ausgabe des Ergebnisses
    cout << »upper = » << upper << »\n«;
    cout << »n     = » << n     << »\n«;
    cout << »lower = » << lower << »\n«;

    return 0;
}
```

Ausgabe:

```
darüber  = 0
n        = 0
darunter = 0

double wird zugewiesen
darüber  = 1076494336
n        = 0
darunter = 0
Press any key to continue
```

Die ersten drei Zeilen in `main( )` deklarieren drei `int`-Variablen auf die bekannte Weise. Wir nehmen hier an, dass diese drei Variablen hintereinander im Speicher angeordnet werden.

Die drei folgenden Zeilen geben die Werte der drei Variablen aus. Es ist nicht überraschend, dass alle Variablen Null sind. Die Zuweisung `*pD = 13.0`; speichert den `double`-Wert 13.0 in der `int`-Variablen `n`. Die drei Ausgabezeilen zeigen die Werte der Variablen nach dieser Zuweisung.

Nachdem der double-Wert 13.0 der `int`-Variablen `n` zugewiesen wurde, ist `n` unverändert, aber die benachbarte Variable `upper` ist mit »Müll« gefüllt.

Das Folgende castet einen Zeiger von einem Typ in einen anderen:
`double* pD = (double*)&n;`
Hierbei ist `&n` ***vom Typ*** `int*`***, wohingegen die Variable*** `pD` ***vom Typ*** `double*` ***ist. Der Cast*** `(double*)` ***ändert den Typ des Werts*** `&n` ***in den Wert von*** `pD`***, in gleicher Weise castet***
`double d = (double)n;`
den `int`***-Wert in*** `n` ***in einen*** `double`***-Wert.***

13.4 Übergabe von Zeigern an Funktionen

Einer der Nutzen von Zeigervariablen ist die Übergabe von Argumenten an Funktionen. Um zu verstehen, warum das wichtig ist, müssen Sie verstehen, wie Argumente an Funktionen übergeben werden.

13.4.1 Wertübergabe

Sie werden bemerkt haben, dass es normalerweise unmöglich ist, den Wert einer Variablen innerhalb der Funktion zu ändern, an die die Variable übergeben wurde. Betrachten Sie das folgende Codesegment:

```
void fn(int nArg)
{
    nArg = 10;
    // der Wert von nArg ist hier 10
}

void parent(void)
{
    int n1 = 0;
    fn(n1);
    // der Wert von n1 ist hier 0
}
```

Die Funktion `parent( )` initialisiert die `int`-Variable `n1` mit 0. Der Wert von `n1` wird an `fn( )` übergeben. Bei Eintritt in die Funktion ist `nArg` gleich 10. `fn( )` ändert den Wert von `nArg`, bevor sie zu `parent( )` zurückkehrt. Vielleicht überrascht es Sie, dass der Wert von `n1` bei Rückkehr zu `parent( )` immer noch 0 ist.

Der Grund dafür ist, dass C++ nicht die Variable selbst an die Funktion übergibt. Stattdessen übergibt C++ den Wert, den die Variable zum Zeitpunkt des Aufrufes hat. D.h. der Ausdruck wird ausgewertet, selbst wenn es nur ein Variablenname ist, und das Ergebnis wird übergeben. Den Wert einer Variable an eine Funktion zu übergeben, wird Wertübergabe genannt.

Wenn man locker sagt »übergib die Variable x an die Funktion `fn( )`«, meint man damit eigentlich »übergib den Wert des Ausdrucks x«.

13.4.2 Übergabe von Zeigerwerten

Wie jeder andere elementare Typ, wird ein Zeigerargument als Wert übergeben.

```
void fn(int* pnArg)
{
   *pnArg = 10;
}

void parent(void)
{
    int n = 0;
    fn(&n);       // das übergibt die Adresse von n
                  // der Wert von n ist jetzt 10
}
```

In diesem Fall wird die Adresse von `n` an die Funktion `fn( )` übergeben und nicht der Wert von `n`. Der entscheidende Unterschied ist offensichtlich, wenn Sie die Zuweisung in `fn( )` betrachten.

Lassen Sie uns zu unserem früheren Beispiel zurückkehren. Nehmen Sie an, dass `n` an Adresse 0x102 gespeichert ist. Nicht der Wert 10, sondern der »Wert« 0x106 wird durch den Aufruf `fn(&n)` übergeben. Innerhalb von `fn( )` speichert die Zuweisung `*pnArg = 10` den Wert 10 in der `int`-Variablen, die sich an Adresse 0x102 befindet, wodurch der Wert 0 überschrieben wird. Bei Rückkehr zu `parent( )` ist der Wert von `n` gleich 10, weil `n` nur ein anderer Name für 0x102 ist.

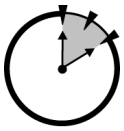
13.4.3 Referenzübergabe

C++ stellt eine kürzere Methode bereit, Variablen zu übergeben, ohne Zeiger verwenden zu müssen. Im folgenden Beispiel wird die Variable `n` als Referenz übergeben. Bei der *Referenzübergabe* gibt die Funktion `parent( )` eine Referenz auf die Variable anstelle ihres Wertes. Referenz ist ein anderes Wort für Adresse.

```
void fn(int& nArg)
{
    nArg = 10;
}

void parent(void)
{
    int n = 0;
    fn(n);        // Übergabe von n als Referenz
                  // hier hat n den Wert 10
}
```

In diesem Fall wird eine Referenz auf `n` an `fn( )` übergeben und nicht der Wert von `n`. Die Funktion `fn( )` speichert den Wert 10 an der `int`-Stelle, die durch `nArg` referenziert wird.

Beachten Sie, dass Referenz kein wirklicher Typ ist. Somit ist der volle Funktionsname `fn(int)` ***und nicht*** `fn(int&)`.



10 Min.

13.5 Heap-Speicher

Genauso wie es möglich ist, einen Zeiger an eine Funktion zu übergeben, ist es für eine Funktion möglich, einen Zeiger zurückzugeben. Eine Funktion, die die Adresse von einem `double` zurückgibt, würde wie folgt deklariert:

```
double* fn(void);
```

Man muss allerdings sehr vorsichtig mit der Rückgabe von Zeigern sein. Um zu verstehen warum, müssen Sie etwas mehr über Geltungsbereiche von Variablen wissen.

13.5.1 Geltungsbereich

C++-Variablen haben zusätzlich zu ihrem Wert und ihrem Typ eine Eigenschaft, die als Geltungsbereich bezeichnet wird. Der Geltungsbereich ist der Bereich, in dem die Variable definiert ist. Betrachten Sie den folgenden Codeschnipsel:

```
// die folgende Variable kann von allen
// Funktionen verwendet werden und ist
// so lange definiert, wie das Programm
// läuft (globaler Geltungsbereich)
int nGlobal;

// die folgende Variabale nChild ist nur in der
// Funktion verfügbar und existiert nur so lange,
```

```
// wie C++ die Funktion child( ) ausführt oder eine
// Funktion, die von child( ) aufgerufen wird
// (Funktionsgeltungsbereich)
void child(void)
{
    int nChild;
}

// die folgende Variable nParent hat
// Funktionsgeltungsbereich
void parent(void)
{
    int nParent = 0;
    fn();

    int nLater = 0;
    nParent = nLater;
}

int main(int nArgs, char* pArgs[])
{
    parent();
}
```

Die Ausführung beginnt bei `main( )`. Die Funktion `main( )` ruft sofort `parent( )` auf. Die erste Sache, die der Prozessor in `parent( )` sieht, ist die Deklaration von `nParent`. An dieser Stelle betritt `nParent` seinen Geltungsbereich – d.h. `nParent` ist definiert und steht im Rest der Funktion `parent ( )` zur Verfügung.

Die zweite Anweisung in `parent( )` ist der Aufruf von `child( )`. Auch `child( )` deklariert eine lokale Variable, diesmal `nChild`. Die Variable `nChild` ist innerhalb des Geltungsbereiches von `child( )`. Technisch gesehen ist `nParent` nicht innerhalb des Geltungsbereiches von `child( )`, weil `child( )` keinen Zugriff auf `nParent` hat. Die Variable `nParent` existiert aber weiterhin.

Wenn `child( )` verlassen wird, verlässt die Variable `nChild` ihren Geltungsbereich. `nChild` wird dadurch nicht nur nicht mehr zugreifbar, sondern existiert gar nicht mehr. (Der Speicher, der von `nChild` belegt wurde, wird an einen allgemeinen Pool zurückgegeben, um für andere Dinge verwendet zu werden.)

Wenn `parent( )` mit der Ausführung fortfährt, betritt die Variable `nLater` bei ihrer Deklaration ihren Geltungsbereich. Wenn `parent( )` zu `main( )` zurückkehrt, verlassen beide, `nParent` und `nLater` ihren Geltungsbereich.

Der Programmierer kann Variablen außerhalb jeder Funktion deklarieren. Eine globale Variable ist eine Variable, die außerhalb jeder Funktion deklariert ist. Eine solche Variable bleibt in ihrem Geltungsbereich für die gesamte Dauer des Programms.

Weil `nGlobal` in diesem Beispiel global deklariert wurde, steht diese Variable allen drei Funktionen zur Verfügung und bleibt für die gesamte Dauer des Programms verfügbar.

13.5.2 Das Geltungsbereichsproblem

Das folgende Codesegment kann ohne Fehler kompiliert werden, aber arbeitet nicht korrekt:

```
double* child(void)
{
    double dLocalVariable;
    return &dLocalVariable;
}

void parent(void)
{
    double* pdLocal;
    pdLocal = child();
    *pdLocal = 1.0;
}
```

Das Problem ist, dass `dLocalVariable` nur innerhalb des Geltungsbereiches der Funktion `child( )` definiert ist. Somit existiert die Variable gar nicht mehr, wenn die Adresse von `dLocalVariable` von `child( )` zurückgegeben wird. Der Speicher, der von `dLocalVariable` belegt wurde, wird möglicherweise bereits für etwas anderes verwendet.

Das ist ein häufiger Fehler, weil er sich auf verschiedene Arten einschleichen kann. Unglücklicherweise lässt dieser Fehler das Programm nicht sofort anhalten. In der Tat kann das Programm in den meisten Fällen korrekt weiterarbeiten, so lange, wie der vorher von `dLocalVariable` belegte Speicher nicht anders verwendet wird. Solche sprunghaften Probleme sind am schwierigsten zu lösen.

13.5.3 Die Heap-Lösung

Das Problem mit dem Geltungsbereich tritt auf, weil C++ den lokal definierten Speicher freigibt, bevor der Programmierer fertig ist. Was benötigt wird, ist ein Speicherbereich, der vom Programmierer verwaltet wird. Der Programmierer kann Speicher allozieren, und ihn auch wieder zurückgeben wenn er mag. Solch ein Speicherbereich wird Heap genannt.

Der *Heap* ist ein Segment des Speichers, das explizit vom Programm kontrolliert wird. Heapspeicher wird über das Kommando `new` alloziert, gefolgt von dem Typ des Objektes, das alloziert werden soll. Z.B. alloziert das folgende eine `double`-Variable vom Heap:

```
double* child(void)
{
    double* pdLocalVariable = new double;
    return pdLocalVariable;
}
```

Obwohl die Variable `pdLocalVariable` ihren Gültigkeitsbereich verlässt, wenn die Funktion `child( )` zurückkehrt, passiert dies nicht mit dem Speicher, auf den `pdLocalVariable` verweist.

Eine Speicherstelle, die von `new` zurückgegeben wird, verlässt ihren Gültigkeitsbereich erst, wenn sie explizit an den Heap mittels den Kommandos `delete` zurückgegeben wird.

```
void parent(void)
{
    // child() gibt die Adresse eines Blocks
    // Speicher vom Heap zurück
    double* pdMyDouble = child();
```

```
    // speichere dort einen Wert
    *pdMyDouble = 1.1;

    // ...

    // gib jetzt den Speicher an den Heap zurück
    delete pdMyDouble;
    pdMyDouble = 0;

    // ...
}
```

0 Min.

Hierbei wird der Zeiger, der von `child`4 zurückgegeben wird, zur Speicherung eines double-Wertes verwendet. Nachdem die Funktion mit der Speicherstelle fertig ist, wird sie an den Heap zurückgegeben. Den Zeiger nach dem delete auf 0 zu setzen ist nicht notwendig, aber eine gute Idee. Wenn der Programmierer versehentlich versucht, etwas in `*pdMyDouble` zu speichern, nachdem bereits `delete` ausgeführt wurde, stürzt das Programm sofort ab.

Ein Programm, das sofort abstürzt, wenn ein Fehler aufgetreten ist, ist viel einfacher zu korrigieren, als ein Programm, das im Falle eines Fehlers sprunghaft ist.

Zeiger auf Funktionen

Zusätzlich zu den Zeigern auf elementare Typen ist es möglich, Zeiger auf Funktionen zu deklarieren. Z.B. deklariert die folgende Zeile einen Zeiger auf eine Funktion, die ein `int`-Argument übernimmt und einen `double`-Wert zurückgibt.

```
double (*pFN)(int);
```

Es gibt eine Vielzahl von Einsatzmöglichkeiten solcher Variablen; die Feinheiten der Funktionszeiger gehen jedoch über den Umfang dieses Buches hinaus.

Zusammenfassung

Zeigervariablen sind ein mächtiger, wenn auch gefährlicher Mechanismus, um auf Objekte über ihre Speicheradresse zuzugreifen. Das ist wahrscheinlich das entscheidende Feature, das die Dominanz von C, und später von C++, gegenüber anderen Programmiersprachen erklärt.

- Zeigervariablen werden durch Hinzufügen eines '`*`' zum Variablentyp deklariert. Der Stern kann irgendwo zwischen dem Variablennamen und dem Basistyp stehen. Es ist aber am sinnvollsten, den Stern ans Ende des Variablentyps zu setzen.
- Der Operator `&` liefert die Adresse eines Objektes zurück, während der Operator `*` das Objekt zurückgibt, auf das eine Adressen- oder Zeigervariable verweist.

- Variablentypen wie `int*` sind eigene Variablentypen, und sind nicht äquivalent mit `int`. Die Adressenoperator `&` konvertiert einen Typ wie z.B. `int` in einen Zeigertyp wie z.B. `int*`. Der Operator `*` konvertiert einen Zeigertyp wie z.B. `int*` in den Basistyp, wie z.B. `int`. Ein Zeigertyp kann mit einem gewissen Risiko in einen anderen Zeigertyp konvertiert werden.

Folgende Sitzungen stellen weitere Wege dar, wie Zeiger die Trickkiste von C++ bereichern können.

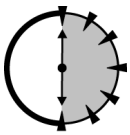
Selbsttest

1. Wenn eine Variable `x` den Wert 10 enthält, und an der Adresse 0x100 gespeichert ist, was ist der Wert von `x`? Was ist der Wert von `&x`? (Siehe »Einführung in Zeigervariablen«)
2. Wenn `x` ein int ist, was ist der Typ von `&x`? (Siehe »Typen von Zeigern«)
3. Warum sollten Sie einen Zeiger an eine Funktion übergeben? (Siehe »Übergabe von Zeigerwerten«)
4. Was ist Heapspeicher und wie erhalten Sie Zugriff darauf? (Siehe »Heapspeicher«)

Mehr über Zeiger

Checkliste

- ☑ Mathematische Operationen auf Zeichenzeigern einführen
- ☑ Die Beziehung von Zeigern und Arrays untersuchen
- ☑ Die Beziehung zur Beschleunigung von Programmen einsetzen
- ☑ Zeigeroperationen auf verschiede Zeigertypen erweitern
- ☑ Die Argumente von `main( )` im C++- Programmtemplate erklären



30 Min.

Die Zeigertypen, die in Sitzung 13 eingeführt wurden, haben einige interessante Operationen ermöglicht. Die Adresse einer Variablen zu speichern, und dann diese Adresse mehr oder weniger wie die Variable selbst zu benutzen, ist schon ein interessanter Partytrick, aber sein Nutzen ist begrenzt, außer bei der permanenten Modifizierung von Variablen, die an eine Funktion übergeben wurden.

Was Zeiger interessant macht, ist die Fähigkeit, mathematische Operationen ausführen zu können. Sicher, weil die Multiplikation zweier Adressen keinen Sinn macht, ist sie auch nicht erlaubt. Dass zwei Adressen jedoch miteinander verglichen werden können und ein Integeroffset zu einer Adresse addiert werden kann, eröffnet interessante Möglichkeiten, die hier untersucht werden.

14.1 Zeiger und Arrays

Einige der Operatoren, die auf Integerzahlen anwendbar sind, können auch auf Zeigertypen angewendet werden. Dieser Abschnitt untersucht deren Auswirkungen auf Zeiger und die Arraytypen, die wir bisher studiert haben.

14.1.1 Operationen auf Zeigern

Tabelle 14-1: Drei Operationen, die für Zeiger definiert sind

Operation	Ergebnis	Bedeutung
pointer + offset	Zeiger	berechne die Adresse offset viele Einträge von Zeiger pointer entfernt
pointer – offset	Zeiger	das Gegenteil zur Addition
pointer2 – pointer1	Offset	berechne den Abstand zwischen den Zeigern pointer2 und pointer1

(Obwohl nicht in Tabelle 14-1 aufgelistet, sind die abgeleiteten Operatoren, wie z.B. `+=offset` und `pointer++` auch als Variation der Addition definiert.)

Das einfache Speichermodell, das in Sitzung 13 zur Erklärung des Zeigerkonzeptes verwendet wurde, ist auch hier nützlich, um die Wirkungsweise der Operatoren zu erklären. Betrachten Sie ein Array von 32 1-Bit-Zeichen, das wir `cArray` nennen wollen. Wenn das erste Byte des Array an Adresse 0x110 gespeichert wird, dann würde das Array den Speicher von 0x110 bis 0x12f belegen. Das Element `cArray[0]` befindet sich an Adresse 0x110, `cArray[1]` an Adresse 0x111, `cArray[2]` an Adresse 0x112 usw.

Nehmen Sie nun an, dass sich ein Zeiger `ptr` an der Adresse 0x102 befindet. Nachdem die folgende Anweisung

```
ptr = &cArray[0];
```

ausgeführt wurde, enthält `ptr` den Wert 0x110. Dies wird in Abbildung 14.1 gezeigt.

Die Addition einer Integerzahl als Offset zum Zeiger ist so definiert, dass die Beziehungen in Tabelle 14-2 wahr sind. Abbildung 14.2 zeigt außerdem, warum die Addition eines Offset `n` zu `ptr` die Adresse des n-ten Elementes von `cArray` berechnet.

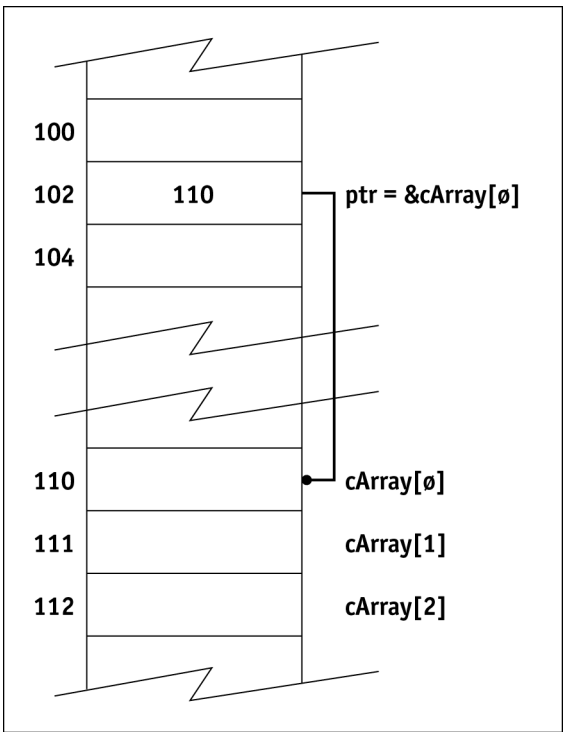


Abbildung 14.1: Nach der Zuweisung ptr = &cArray[0] zeigt der Zeiger ptr auf den Anfang des Arrays cArray.

Tabelle 14-2: Zeigeroffsets und Arrays

Offset	Ergebnis	Entspricht ...
+0	0x110	cArray[0]
+1	0x111	cArray[1]
+2	0x112	cArray[2]
...	...	...
+n	0x110+n	cArray[n]

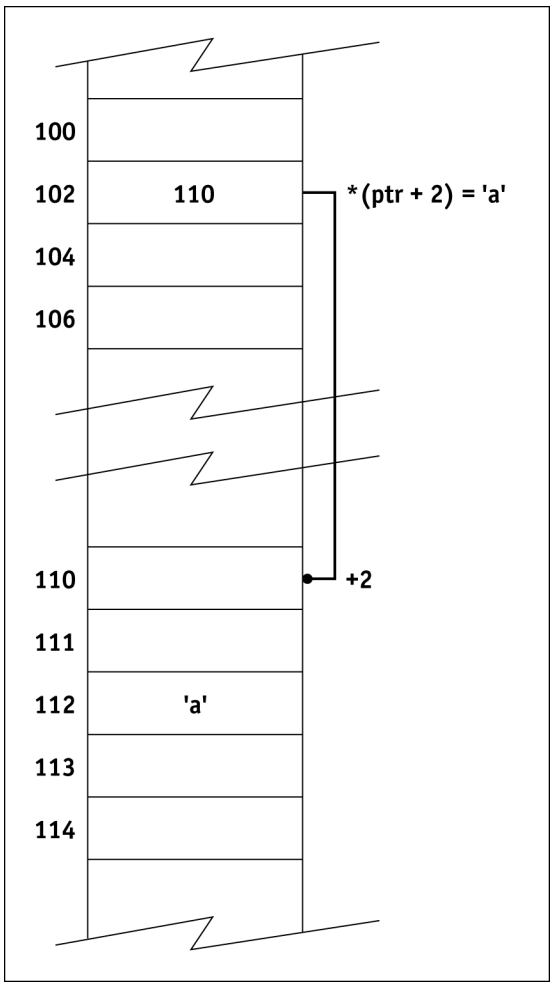


Abbildung 14.2: Der Ausdruck ptr + i hat als Wert die Adresse von cArray[i].

Wenn also

```
char* prt = &cArray[0];
```

gegeben ist, so entspricht

```
*(ptr + n)
```

dem Array-Element

```
cArray[n].
```

Weil `*` ***eine höhere Priorität hat als die Addition, addiert*** `*ptr + n` ***zum Zeiger*** `ptr n` ***Zeichen. Die Klammern werden benötigt, um zu erzwingen, dass die Addition vor dem Operator*** `*` ***ausgeführt wird. Der Ausdruck*** `*(prt + n)` ***greift auf das Zeichen zu, das von*** `ptr` ***aus*** `n` ***Zeichen weiter steht.***

In der Tat ist der Zusammenhang der beiden Ausdrücke so stark, dass C++ `array[n]` als nicht mehr, als nur eine vereinfachte Version von `*(ptr + n)` ansieht, wobei `ptr` auf das erste Element von `array` zeigt.

```
array[n] -> interpretiert C++ als -> *(&array[0] + n)
```

Um die Assoziation zu vervollständigen, verwendet C++ eine weitere Kurzform. Gegeben

```
char cArray[20];
```

dann ist

```
carray == &cArray[0]
```

d.h., der Name des Arrays ohne einen Index repräsentiert die Adresse des Array selber. Wir können also die Assoziation weiter vereinfachen:

```
array[n] -> interpretiert C++ als -> *(array + n)
```

Das ist eine mächtige Aussage. Z.B. könnte die Funktion `displayArray( )` aus Sitzung 11, die den Inhalt eines `int`-Array ausgibt, so geschrieben werden:

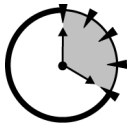
```
// displayArray - zeige die Elemente eines Array
//                der Länge nSize
void displayArray(int nArray[], int nSize)
{
    cout << »Der Wert des Array ist:\n«;

    // zeige auf das erste Element von nArray
    int* pArray = nArray;
    while(nSize—)
    {
        cout.width(3);

        // gib Integer aus, worauf pArray zeigt ...
        cout << i << »: » << *pArray << »\n«;

        // ... und bewege Zeiger zum nächsten
        // Element von nArray
        pArray++;
    }
    cout << »\n«;
}
```

Die neue Funktion `displayArray( )` beginnt damit, einen Zeiger `pArray` auf das `int-Array` zu erzeugen, der auf das erste Element von `nArray` zeigt.

Gemäß unserer Konvention weist das `p` auf einen Zeiger hin.

Die Funktion durchläuft dann eine Schleife über jedes Element des Array (wobei `nSize` als die Anzahl der Elemente im Array verwendet wird). In jedem Schleifendurchlauf gibt die Funktion `displayArray( )` die entsprechende Integerzahl aus, d.h. das `int`-Element, auf das `pArray` zeigt, bevor der Zeiger zum nächsten Element in `nArray` inkrementiert wird.

Diese Verwendung von Zeigern zum Zugriff auf Arrays wird nirgendwo sonst so häufig verwendet wie bei Zeichenarrays.



20 Min.

14.1.2 Zeichenarrays

Sitzung 11 hat auch erklärt, dass C++ Null-terminierte Zeichenarrays wie einen Datentyp verwendet. C++-Programmierer verwenden oft Zeichenzeiger, um solche Zeichenketten zu manipulieren. Der folgende Beispielcode vergleicht diese Technik mit der früheren Technik der Array-Indizierung.

Zeiger und Array-basierte Manipulation von Zeichenketten

Die Funktion `concatString( )` wurde im Beispiel Concatenate in Sitzung 11 deklariert.

```
void concatString(char szTarget[], char szSource[]);
```

Die Prototypdeklaration beschreibt die Typen der Argumente, die die Funktion entgegennimmt, sowie den Rückgabetyp. Diese Deklaration sieht wie die Definition der Funktion aus, nur ohne Body.

Um die Null am Ende des Array `szTarget` zu finden, iteriert die Funktion `concatString( )` durch die Zeichenkette `szTarget` mit der folgenden `while`-Schleife:

```
void concatString(char szTarget[], char szSource[])
{
    // finde das Ende der ersten Zeichenkette
    int nTargetIndex = 0;
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // ...
```

Unter Verwendung der Beziehung zwischen Zeigern und Arrays könnte die Funktion `concatString( )` auch folgenden Prototyp besitzen:

```
void concatString(char* pszTarget, char* pszSource);
```

Das `z` weist auf eine durch 0 (null) abgeschlossene Zeichenkette hin.

Die Zeigerversion von `concatString( )`, die im Programm ConcatenatePtr enthalten ist, sieht dann wie folgt aus:

```
void concatString(char* pszTarget, char* pszSource)
{
    // finde das Ende der ersten Zeichenkette
    while(*pszTarget)
    {
        pszTarget++;
    }

    // ...
```

Die `while`-Schleife in der Arrayversion von `concatString( )` wurde verlassen, sobald `szTarget[nTargetIndex]` gleich 0 war. Diese Version nun iteriert durch das Array, indem `pszTarget` in jedem Schleifendurchlauf inkrementiert wird, bis das Zeichen, auf das `pszTarget` zeigt, gleich null ist.

Der Ausdruck `ptr++` ***ist eine Kurzform von*** `ptr = ptr + 1`.

Wenn die `while`-Schleife verlassen wurde, zeigt `pszTarget` auf das Nullzeichen am Ende der Zeichenkette `szTarget`.

Es ist nicht mehr richtig zu sagen »das Array, auf das `pszTarget` ***zeigt«, weil*** `pszTarget` ***nicht mehr auf den Anfang des Array zeigt.***

Das vollständige Beispiel concatString( )
Und hier ist das vollständige Programm ConcatenatePtr:

```
 1.  // ConcatenatePtr - verbindet zwei Zeichenketten
 2.  //                 mit » - « in der Mitte unter
 3.  //                 Verwendung von Zeigern statt
 4.  //                 Arrayindizes
 5.  #include <stdio.h>
 6.  #include <iostream.h>
 7.
 8.  void concatString(char* pszTarget, char* pszSource);
 9.
10.  int main(int nArg, char* pszArgs[])
11.  {
12.      // lies erste Zeichenkette ...
13.      char szString1[256];
14.      cout << »Zeichenkette #1:«;
15.      cin.getline(szString1, 128);
16.
17.      // ... nun die zweite Zeichenkette ...
18.      char szString2[128];
19.      cout << »Zeichenkette #2:«;
20.      cin.getline(szString2, 128);
21.
22.      // ... füge » - « an die erste ...
23.      concatString(szString1, » - «);
24.
25.      // ... füge jetzt die zweite an ...
26.      concatString(szString1, szString2);
27.
28.      // ... und stelle das Ergebnis dar
29.      cout << »\n« << szString1 << »\n«;
30.
31.      return 0;
32.  }
33.
34.  // concatString - fügt *pszSource an das Ende
35.  //               von *pszTarget an
36.  void concatString(char* pszTarget, char* pszSource)
```

```
37.  {
38.      // finde das Ende der ersten Zeichenkette
39.      while(*pszTarget)
40.      {
41.          pszTarget++;
42.      }
43.
44.      // hänge die zweite ans Ende der ersten
45.      // (kopiere auch die Null des Quellarrays -
46.      // dadurch wird das Zielarray mit einem
47.      // Nullzeichen abgeschlossen)
48.      while(*pszTarget++ = *pszSource++)
49.      {
50.      }
51.
```

Die Funktion `main( )` des Programms unterscheidet sich nicht von ihrer Array-basierten Cousine. Die Funktion `concatString( )` ist jedoch signifikant verschieden.

Wie bereits erwähnt, basiert die äquivalente Deklaration von `concatString( )` nun auf Zeigern vom Typ `char*`. Zusätzlich sucht die erste `while`-Schleife innerhalb von `concatString( )` nach dem abschließenden Null-Zeichen am Ende des Arrays `pszTarget`.

Die extrem kompakte Schleife, die dann folgt, kopiert das Array `pszSource` an das Ende des Array `pszTarget`. Die `while`-Klausel macht die ganze Arbeit, indem sie die folgenden Dinge ausführt:

1. Hole das Zeichen, auf das `pszSource` zeigt.
2. Inkrementiere `pszSource` zum nächsten Zeichen.
3. Speichere das Zeichen an der durch `pszTarget` gegebenen Position.
4. Inkrementiere `pszTarget` zum nächsten Zeichen.
5. Führe den Body der Schleife so lange aus, bis das Zeichen 0 (null) ist.

Nachdem der leere Rumpf der `while`-Schleife verlassen wurde, wird die Kontrolle an die `while ( )`-Klausel selber zurückgegeben. Die Schleife wird so lange wiederholt, bis das Zeichen, das nach `*pszTarget` kopiert wurde, gleich dem Nullzeichen ist.

Warum Arrayzeiger?

Die manchmal kryptische Natur von Zeiger-basierter Manipulation von Zeichenketten kann den Leser schnell zur Frage »warum?« führen. D.h. welchen Vorteil bietet die `char*`-basierte Zeigerversion von `concatString( )` gegenüber der doch leichter lesbaren Indexversion?

Die Zeigerversion von `concatString( )` kommt in C++-Programmen häufiger vor als die Array-Version aus Sitzung 11.

Die Antwort ist teilweise historisch und teilweise menschlicher Natur. So kompliziert sie auch für den menschlichen Leser aussehen mag, kann eine Anweisung wie in Zeile 48 in eine unglaublich kleine Anzahl von Maschineninstruktionen überführt werden. Ältere Computerprozessoren waren nicht so schnell wie die heutigen. Als C, der Vorgänger von C++, vor etwa 30 Jahren entwickelt wurde, war es toll, einige Maschineninstruktionen einsparen zu können. Das gab C einen großen Vorteil

gegenüber anderen Sprachen aus dieser Zeit, insbesondere gegenüber FORTRAN, die keine Zeigerarithmetik enthielt.

Außerdem mögen es die Programmierer, clevere Programme zu schreiben, damit die Sache nicht langweilig wird. Wenn C++-Programmierer erst einmal gelernt haben, wie man kompakte und kryptische, aber effiziente Anweisungen schreibt, gibt es kein Mittel, sie wieder zur Suche in Arrays mittels Index zurückzubringen.

Erzeugen Sie keine komplexen C++-Ausdrücke mit der Absicht, effizienteren Code zu erzeugen. Es gibt keinen offensichtlichen Zusammenhang zwischen der Anzahl der C++-Anweisungen und der Anzahl der Maschineninstruktionen. Z.B. könnten die beiden folgenden Ausdrücke die gleiche Anzahl von Maschineninstruktionen erzeugen:

```
*pszArray1++ = '\0';

*pszArray2 = '0';
pszArray2 = pszArray2 + 1;
```

Früher, als die Compiler noch einfacher gebaut waren, hätte die erste Version sicherlich weniger Instruktionen erzeugt.

14.1.3 Operationen auf unterschiedlichen Zeigertypen

Die beiden Beispiele von Zeigermanipulationen, die bisher gezeigt wurden, `concatString(char*, char*)` und `displayArray(int*)`, unterscheiden sich grundlegend in zwei Punkten:

Es ist für Sie nicht sehr schwer, sich selbst davon zu überzeugen, dass `szTarget + n` auf `szTarget[n]` zeigt, wenn Sie berücksichtigen, dass jedes Zeichen in `szTarget` ein einziges Byte belegt. Wenn also `szTarget` an Adresse 0x100 gespeichert ist, dann steht das sechste Element an Adresse 0x105 (0x100 + 5 ist gleich 0x105).

Weil C++-Arrays bei 0 zu zählen beginnen, ist `szTarget[5]` ***das sechste Element des Array.***

Es ist nicht offensichtlich, dass Zeigeraddition auch für `nArray` funktioniert, da jedes Element von `nArray` ein `int` ist und damit 4 Bytes belegt. Wenn das erste Element von `nArray` an Adresse 0x100 steht, dann steht das sechste Element an Adresse 0x114 (0x100 + (5 * 4) = 0x114).

Glücklicherweise zeigt in C++ `array + n` auf das Element `array[n]`, unabhängig davon, wie groß ein einzelnes Element von `array` ist.

Eine gute Parallele bieten Häuserblocks in einer Stadt. Wenn alle Adressen in jeder Straße fortlaufend ohne Lücken, nummeriert wären, dann wäre die Hausnummer 1605 das sechste Haus in Block 1600. Um den Postboten nicht zu sehr zu verwirren, wird diese Beziehung eingehalten, unabhängig von der Größe der Häuser.

14.1.4 Unterschiede zwischen Zeigern und Arrays

Außer den äquivalenten Typen, gibt es einige Unterschiede zwischen einem Array und einem Zeiger. Zum einen alloziert ein Array Speicher für die Daten, ein Zeiger tut das nicht:

```
void arrayVsPointer()
{
    // alloziere Speicher für 128 Zeichen
    char cArray[128];

    // alloziere Speicher für einen Zeiger
    char* pArray;
}
```

Hier belegt `cArray` 128 Bytes, das ist der Speicher, der für 128 Zeichen benötigt wird. `pArray` belegt nur 4 Bytes, das ist der Speicher, der für einen Zeiger benötigt wird.

Die folgende Funktion funktioniert nicht:

```
void arrayVsPointer()
{
    // greife auf Elemente mit Array zu
    char cArray[128];
    cArray[10] = '0';
    *(cArray + 10) = '0';

    // greife auf ein 'Element' des Arrays
    // zu, das nicht existiert
    char* pArray;
    pArray[10] = '0';
    *(pArray + 10) = '0';
}
```

Der Ausdruck `cArray[10]` und `*(cArray + 10)` sind äquivalent und zulässig. Die beiden Ausdrücke, die `pArray` enthalten, machen keinen Sinn. Während sie in C++ zulässig sind, enthält das nicht initialisierte `pArray` einen zufälligen Wert. Somit versucht das zweite Anweisungspaar ein Nullzeichen irgendwo in den Speicher zu schreiben.

Diese Art Fehler wird im Allgemeinen von der CPU abgefangen und resultiert dann im gefürchteten Fehler einer Segmentverletzung, den Sie hin und wieder bei Ihren Lieblingsprogrammen antreffen.

Ein zweiter Unterschied ist, dass `cArray` eine Konstante ist, was auf `pArray` nicht zutrifft. Somit arbeitet die folgende `for`-Schleife, die das `Array cArray` initialisieren soll, nicht korrekt:

```
void arrayVsPointer()
{
    char cArray[10];
    for (int i = 0; i < 10; i++)
    {
        *cArray = '\0';    // das macht Sinn ...
        cArray++;          // ... das nicht
    }
}
```

Der Ausdruck `cArray++` macht nicht mehr Sinn als `10++`. Die korrekte Version sieht so aus:

```
void arrayVsPointer()
{
    char cArray[10];
    char* pArray = cArray;
    for (int i = 0; i < 10; i++)
    {
        *pArray = '\0';    // das funktioniert
        pArray++;
    }
}
```

14.2 Argumente eines Programms

Arrays von Zeigern sind ein anderer Typ Array, der von besonderem Interesse ist. Dieser Abschnitt untersucht, wie Sie diese Arrays einsetzen können, um Ihre Programme einfacher zu machen.

14.2.1 Arrays von Zeigern

Weil Arrays Daten beliebigen Typs enthalten können, ist es möglich, Zeiger in Arrays zu speichern. Das Folgende deklariert ein Array von Zeigern auf `int`:

```
int* pnInts[10];
```

Gegeben die obige Deklaration, ist `pnInts[0]` ein Zeiger auf einen `int`-Wert. Somit ist das Folgende wahr:

```
void fn()
{
    int n1;
    int* pnInts[3];
    pnInts[0] = &n1;
    *pnInts[0] = 1;
}
```

oder

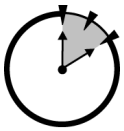
```
void fn()
{
    int n1, n2, n3;
    int* pnInts[3] = {&n1, &n2, &n3};
    for (int i = 0; i < 3; i++)
    {
        *pnInts[i] = 0;
    }
}
```

oder sogar

```
void fn()
{
    int* pnInts[3] = {(new int),
                      (new int),
                      (new int)};
    for (int i = 0; i < 3; i++)
    {
        *pnInts[i] = 0;
    }
}
```

Das Letztere deklariert drei `int`-Objekte vom Heap.

Der häufigste Einsatz von Zeigerarrays ist das Anlegen eines Array von Zeichenketten. Die folgenden zwei Beispiele zeigen, warum Arrays von Zeichenketten nützlich sind.



10 Min.

14.2.2 Arrays von Zeichenketten

Betrachten Sie eine Funktion, die den Namen eines Monats zurückgibt, der zu einer Integerzahl gehört. Wenn das Programm z.B. den Wert 1 erhält, gibt es die Zeichenkette »Januar« zurück.

Die Funktion könnte wie folgt geschrieben werden:

```
// int2month() - gib den Namen des Monats zurück
char* int2month(int nMonth)
{
    char* pszReturnValue;

    switch(nMonth)
    {
        case 1: pszReturnValue = »Januar«;
                break;
        case 2: pszReturnValue = »Februar«;
                break;
        case 3: pszReturnValue = »März«;
                break;
        // ... usw ...
        default: pszReturnValue = »invalid«;
    }
    return pszReturnValue;
}
```

Wenn 1 übergeben wird, geht die Kontrolle an die erste `case`-Anweisung über und die Funktion würde einen Zeiger auf die Zeichenkette »Januar« zurückgeben; wenn eine 2 übergeben wird, kommt »Februar« zurück usw.

Das* `switch( )`*-Kontrollkommando ist vergleichbar mit einer Folge von* `if`*-Anweisungen.

Eine elegantere Lösung nutzt den Integerwert als Index für ein Array von Zeigern auf die Monatsnamen. Praktisch sieht das so aus:

```
// int2month() - gib den Namen des Monats zurück
char* int2month(int nMonth)
{
    // überprüfe den Wert auf Gültigkeit
    if (nMonth < 1 || nMonth > 12)
    {
        return »ungültig«;
    }

    // nMonth ist gültig - gib Monatsnamen zurück
```

```
    char* pszMonths[] = {»invalid«,
                         »Januar«,
                         »Februar«,
                         »März«,
                         »April«,
                         »Mai«,
                         »Juni«,
                         »Juli«,
                         »August«,
                         »September«,
                         »Oktober«,
                         »November«,
                         »Dezember«};
    return pszMonths[nMonth];
}
```

Hierbei überprüft die Funktion `int2month( )` zuerst, ob der Wert von `nMonth` zwischen 1 und 12 ist (die `default`-Klausel der `switch`-Anweisung hat das für uns im vorhergehenden Beispiel erledigt). Wenn `nMonth` zulässig ist, benutzt die Funktion diesen Wert als Offset für das Array, das die Monatsnamen enthält.

14.2.3 Die Argumente von main( )

Sie haben bereits einen anderen Einsatz eines Zeigerarray auf Zeichenketten gesehen: Die Argumente der Funktion `main( )`.

Die Argumente eines Programms sind die Zeichenketten, die beim Aufruf hinter dem Programmnamen angegeben werden. Nehmen Sie z.B. an, dass ich das folgende Kommando hinter dem MS-DOS-Prompt eingegeben habe:

```
MyProgram file.txt /w
```

MS-DOS führt das Programm, das in der Datei `MyProgram.exe` enthalten ist, aus und übergibt ihm die Argumente `file.txt` und `/w`.

Der Nutzen des Begriffs Argument ist ein wenig verwirrend. Die Argumente eines Programms und die Argumente einer C++-Funktion folgen einer unterschiedlichen Syntax, aber die Bedeutung ist die gleiche.

Betrachten Sie das folgende Beispielprogramm:

```
// PrintArgs - schreibt die Argumente des Programms
//             in die Standardausgabe
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // gib Kopfzeile aus
    cout << »Argument von « << pszArgs[0] << »\n«;
```

```
    // nun schreibe die Argumente raus
    for (int i = 1; i < nArg; i++)
    {
        cout << i << »:« << pszArgs[1] << »\n«;
    }

    // das war es
    cout << »Das war es\n«;
    return 0;
}
```

Wie immer akzeptiert die Funktion `main( )` zwei Argumente. Das erste ist ein `int` und trägt den Namen `nArgs`. Diese Variable enthält die Anzahl der Argumente, die an das Programm übergeben wurden. Das zweite Argument ist ein Array von Zeigern vom Typ `char*`, das ich `pszArgs` genannt habe. Jedes dieser `char*`-Elemente zeigt auf ein Argument, das dem Programm übergeben wurde.

Betrachten Sie das Programm `PrintArgs`. Wenn ich das Programm aufrufe mit

```
PrintArgs arg1 arg2 arg3 /w
```

von der Kommandozeile eines MS-DOS-Fensters aus, wäre `nArgs` gleich 5 (eins für jedes Argument). Das erste Argument ist der Name des Programms selber. Somit zeigt `psArgs[0]` auf `PrintArgs`. Die restlichen Elemente in `pzArgs` zeigen auf die Programmargumente. Das Element `pszArgs[1]` zeigt auf `arg1`, `pszArgs[2]` zeigt auf `arg2` usw. Weil MS-DOS `/w` keine besondere Bedeutung beimisst, wird diese Zeichenkette in gleicher Weise als ein Argument an das Programm übergeben.

Das Gleiche gilt nicht für die Richtungszeichen »<«, »>« und »|«. Diese haben unter MS-DOS eine besondere Bedeutung und werden nicht als Argument an das Programm übergeben.

Es gibt verschiedene Wege, Argumente an eine Funktion zu übergeben. Der einfachste Weg ist, das Programm vom MS-DOS-Prompt aus aufzurufen. Beide Debugger, von Visual C++ und von GNU C++, stellen einen Mechanismus bereit, um Argumente während des Debuggens zu übergeben.

In Visual C++, wählen Sie das Debug-Feld in der Dialog-Box »Project Settings« aus. Geben Sie Ihre Argumente in das Eingabefenster »Program Arguments« ein wie in Abbildung 14.3 zu sehen ist. Das nächste Mal, wenn Sie Ihr Programm starten, übergibt Visual C++ diese Argumente.

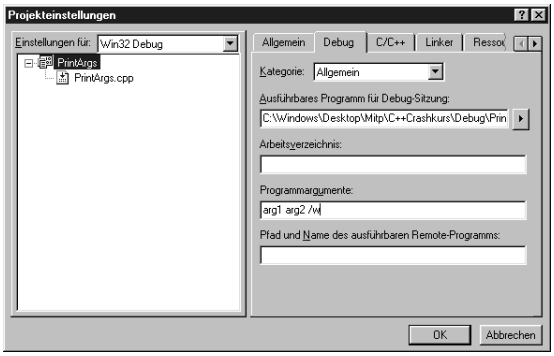


Abbildung 14.3: Visual C++ verwendet Project Settings zur Übergabe von Argumenten an das Programm während des Debuggens.

In `rhide` wählen Sie »Argumente...« im Menü »Start«. Geben sie die Argumente im Fenster ein. Dies ist in Abbildung 14.4 zu sehen.

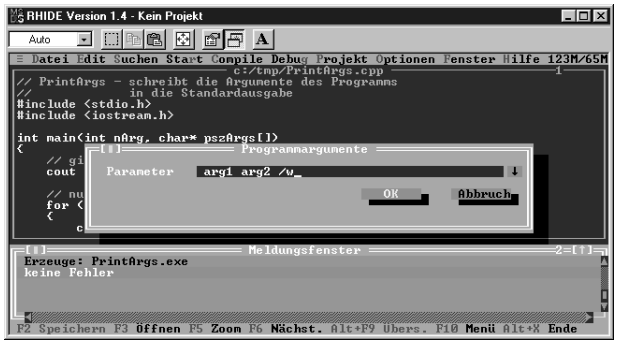


Abbildung 14.4: In `rhide` ***stehen die Programmargumente im Start-Menü.***

0 Min.

Zusammenfassung

Alle Programmiersprachen basieren die Indizierung von Arrays auf einfachen mathematischen Operationen auf Zeigern. Durch die Möglichkeit für den Programmierer, auf diese Art Operation direkt zuzugreifen, gibt C++ dem Programmierer eine große semantische Freiheit. Der C++-Programmierer kann die Beziehung zwischen der Manipulation von Arrays und Zeigern untersuchen und zu seinem Vorteil nutzen.

In dieser Sitzung haben wir gesehen, dass

- die Indizierung auf Arrays einfache mathematische Operationen auf Zeigern beinhaltet. C++ ist praktisch einzigartig darin, dass der Programmierer diese Operationen selber ausführen kann.
- Zeigeroperationen auf Zeichenarrays boten das größte Potenzial zur Leistungssteigerung bei den frühen C- und C++-Compilern. Ob das immer noch der Fall ist, darüber lässt sich streiten. Jedenfalls sind Zeichenzeiger Teil des täglichen Lebens geworden.

- C++ passt die Zeigerarithmetik an die unterschiedliche Größe der Objekte an, auf die Zeiger verweisen. Somit vergrößert die Inkrementierung eines `char`-Zeigers dessen Wert um 1, während die Inkrementierung eines `double`-Zeigers zu einer Vergrößerung seines Wertes um 8 führt. Die Inkrementierung eines Zeigers auf Klassenobjekte kann zu einer Vergrößerung von Hunderten von Bytes führen.
- Arrays von Zeigern können signifikant die Effizient eines Programms erhöhen, das einen `int`-Wert in eine Konstante eines anderen Typs verwandelt, wie z.B. eine Zeichenkette oder ein Bitfeld.
- Argumente eines Programms werden an die Funktion `main( )` als Array von Zeigern auf Zeichenketten übergeben.

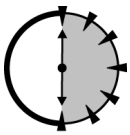
Selbsttest

1. Wenn das erste Element eines Array von Zeichen `c[]` an Adresse 0x100 steht, was ist die Adresse von `c[2]`? (Siehe »Operationen auf Zeigern«)
2. Was ist das Indexäquivalent zum Zeigerausdruck `*(c + 2)`? (Siehe »Zeiger und Array-basierte Manipulation von Zeichenketten«)
3. Was ist der Sinn der beiden Argumente von `main( )`? (Siehe »Die Argumente von main( )«)

Zeiger auf Objekte

Checkliste

- ☑ Zeiger auf Objekte deklarieren und verwenden
- ☑ Objekte durch Zeiger übergeben
- ☑ Objekte vom Heap allozieren
- ☑ Verkettete Listen erzeugen und manipulieren
- ☑ Verkettete Listen von Objekten und Objektarrays vergleichen



30 Min.

Sitzung 12 demonstrierte, wie Arrays und Klassenstrukturen in Arrays von Objekten kombiniert werden können, um eine Reihe von Problemen zu lösen. In gleicher Weise löst die Einführung von Zeigern auf Objekte einige Probleme, die von Objektarrays nicht so leicht gelöst werden können.

15.1 Zeiger auf Objekte

Ein Zeiger auf eine vom Programmierer definierte Struktur arbeitet im Wesentlichen so, wie Zeiger auf die elementaren Typen:

```
int* pInt;
class MyClass
{
  public:
    int  n1;
    char c2;
};
MyClass  mc;
MyClass* pMS = &mc;
```

Der Typ von `pMS` ***ist »Zeiger auf MyClass«, was auch als*** `MyClass*` ***ausgedrückt werden kann.***

Auf Elemente eines solchen Objektes kann wie folgt zugegriffen werden:

```
(*pMS).n1 = 1;
(*pMS).c2 = '\0';
```

In Wörtern besagt der erste Ausdruck »weise 1 dem Element `n1` des MS-Objektes zu, auf das `pMS` verweist.«

Die Klammern sind notwendig, weil ».« eine höhere Priorität hat als »*«. Der Ausdruck `*mc.pN1` ***bedeutet »die Integerzahl, auf die das*** `pN1`***-Element des Objektes*** `mc` ***verweist«.***

Genauso wie C++ eine Kurzform für den Gebrauch von Arrays bereitstellt, definiert C++ bequemere Operatoren, um auf die Elemente eines Objektes zuzugreifen. Der Operator -> ist wie folgt definiert:

```
(*pMS).n1 ist äquivalent zu pMS->n1
```

Der Pfeiloperator wird fast ausschließlich verwendet, weil das so leichter zu lesen ist. Die beiden Formen sind jedoch völlig äquivalent.

15.1.1 Übergabe von Objekten

Ein Zeiger auf ein Klassenobjekt kann an eine Funktion in der gleichen Weise übergeben werden wie einfache Zeigertypen.

```
// PassObjectPtr - demonstriert Funktionen, die einen
//                 Zeiger auf ein Objekt erwarten
#include <stdio.h>
#include <iostream.h>

// MyClass - eine Testklasse ohne Bedeutung
class MyClass
{
  public:
    int n1;
    int n2;
};

// myFunc - Version mit Wertübergabe
void myFunc(MyClass mc)
{
    cout << »In myFunc(MyClass)\n«;
    mc.n1 = 1;
    mc.n2 = 2;
}
```

```
// myFunc - Version mit Zeigerübergabe
void myFunc(MyClass* pMS)
{
    cout << »In myFunc(MyClass*)\n«;
    pMS->n1 = 1;
    pMS->n2 = 2;
}

int main(int nArg, char* pszArgs[])
{
    // Definiere ein Dummy-Object
    MyClass mc = {0, 0};
    cout << »Anfangswert = \n«;
    cout << »n1 = » << mc.n1 << »\n«;

    // übergib als Wert
    myFunc(mc);
    cout << »Ergebnis = \n«;
    cout << »n1 = » << mc.n1 << »\n«;

    // übergib als Zeiger
    myFunc(&mc);
    cout << »Ergebnis = \n«;
    cout << »n1 = » << mc.n1 << »\n«;
    return 0;
}
```

Das Hauptprogramm erzeugt ein Objekt aus der Klasse `MyClass`. Das Objekt wird zuerst an die Funktion `myFunc(MyClass)` übergeben, und dann wird die Adresse des Objektes an die Funktion `myFunc(MyClass*)` übergeben. Beide Funktionen ändern den Wert des Objektes – nur die Änderungen, die innerhalb von `myFunc(MyClass*)` durchgeführt wurden, bleiben erhalten.

Im Aufruf von `myFunc(MyClass)` macht C++ eine Kopie des Objektes. Änderungen am Objekt mc in dieser Funktion bleiben in `main( )` nicht erhalten. Der Aufruf von `myFunc(MyClass*)` übergibt die Adresse auf das ursprüngliche Objekt in `main( )`. Das Objekt enthält alle gemachten Änderungen, wenn die Kontrolle an `main( )` zurückgegeben wird.

Dieser Vergleich von Kopie und Original ist das Gleiche, wie der Vergleich der beiden Funktionen `fn(int)` und `fn(int*)`.

Abgesehen vom Erhalt von Änderungen kann die Übergabe eines 4-Byte-Zeigers wesentlich effizienter sein, als die Kopie eines Objektes.

15.1.2 Referenzen

Sie können Referenzen verwenden, um C++ einige Zeigermanipulationen durchführen zu lassen.

```
// myFunc - mc behält Änderungen in
//          aufrufender Funktion
void myFunc(MyClass& mc)
{
    mc.n1 = 1;
    mc.n2 = 2;
}

int main(int nArgs, char* pszArgs[])
{
    MyClass mc;
    myFunc(mc);
    // ...
```

Sie haben dieses Feature bereits gesehen. Das Beispiel `ClassData` ***in Sitzung 12 verwendete eine Referenz auf ein Klassenobjekt im Aufruf*** `getData(NameDataSet&)`, ***um die gelesenen Daten an den Aufrufenden zurückzugeben.***

15.1.3 Rückgabe an den Heap

Man muss sehr vorsichtig sein, keine Referenz auf ein Objekt, das lokal definiert wurde, zurückzugeben:

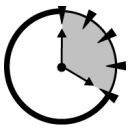
```
MyClass* myFunc()
{
    MyClass  mc;
    MyClass* pMC = &mc;
    return pMC;
}
```

Wenn `myFunc( )` zurückkehrt, verlässt das Objekt mc seinen Gültigkeitsbereich. Der Zeiger, der von `myFunc( )` zurückgegeben wird, ist nicht gültig in der aufrufenden Funktion. (Siehe Sitzung 13 zu Details.)

Das Objekt vom Heap zu allozieren, löst das Problem:

```
MyClass* myFunc()
{
    MyClass* pMC = new MyClass;
    return pMC;
}
```

Der Heap wird verwendet, um Objekte in verschiedenen Situationen zu allozieren.



20 Min.

15.2 Die Datenstruktur Array

Als ein Container von Objekten hat das Array eine Reihe von Vorteilen, insbesondere die Fähigkeit, auf ein Element schnell und effizient zugreifen zu können:

```
MyClass mc[100]; // alloziere Platz für 100 Einträge
mc[n];           // Zugriff auf den n-ten Eintrag
```

Doch das Array hat auch Nachteile:

Arrays haben eine fest Länge. Sie können die Anzahl der Arrayelemente zwar zur Laufzeit bestimmen, aber wenn Sie das Array erst einmal angelegt haben, können Sie seine Größe nicht mehr verändern.

```
void fn(int nSize)
{
    // alloziere ein Objekt, um n Objekte aus
    // der Klasse MyClass zu speichern
    MyClass* pMC = new MyClass[n];

    // die Größe des Arrays ist jetzt fest und
    // kann nicht mehr geändert werden

    // ...
}
```

Zusätzlich muss jeder Eintrag im Array vom gleichen Typ sein. Es ist nicht möglich, Objekte der Klassen `MyClass` und `YourClass` im gleichen Array zu speichern.

Schließlich ist es schwierig, ein Objekt mitten in das Array einzufügen. Um ein Objekt hinzuzufügen oder zu löschen, muss das Programm angrenzende Objekte nach oben oder nach unten kopieren, um eine Lücke zu schaffen oder zu schließen.

Es gibt Alternativen zu Arrays, die diese Einschränkungen nicht besitzen. Die bekannteste ist die verkettete Liste.

15.3 Verkettete Listen

Die verkettete Liste verwendet das gleiche Prinzip wie bei der Übung »wir geben uns die Hände, um die Straße zu überqueren«, als Sie noch ein Kind waren. Jedes Objekt enthält einen Verweis auf das nächste Objekt in der Kette. Der »Lehrer«, auch als Kopfzeiger bekannt, zeigt auf das erste Element der Liste.

Eine verkettete Liste ist wie folgt deklariert:

```
class LinkableClass
{
  public:
    LinkableClass* pNext;

    // andere Elemente der Klasse
};
```

Hierbei zeigt pNext auf den nächsten Eintrag in der Liste. Das sehen Sie in Abbildung 15.1.

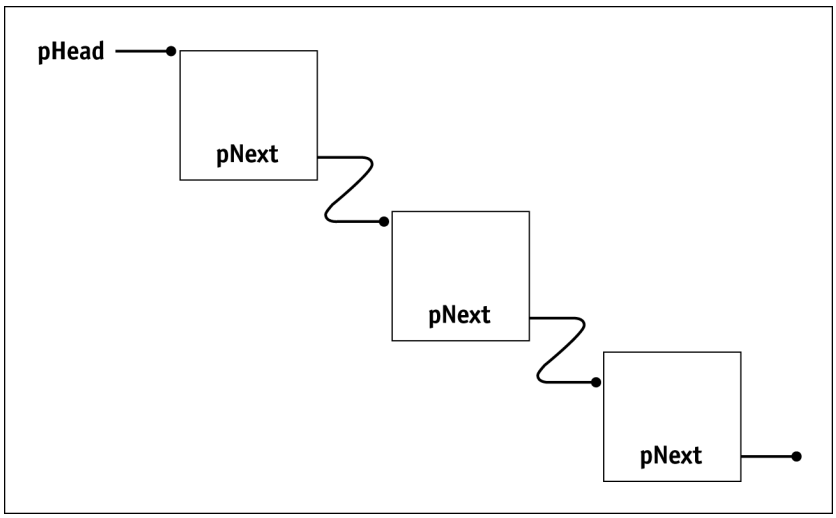


Abbildung 15.1: Eine verkettete Liste besteht aus einer Anzahl von Objekten; jedes Objekt verweist auf das nächste Objekt in der Liste.

Der Kopfzeiger ist einfach ein Zeiger von Typ `LinkableClass*`:

```
LinkableClass* pHead = (LincableClass*)0;
```

Initialisieren Sie jeden Zeiger mit 0, der im Kontext von Zeigern auch als null bezeichnet wird. Dieser Wert wird als Nullzeiger bezeichnet. In jedem Fall wird der Zugriff auf die Adresse 0 immer zum Anhalten des Programms führen.

Der Cast von 0 als Typ int nach `LinkableClass*` ***ist nicht nötig. C++ interpretiert 0 als beliebigen Typ, als eine Art »universeller Zeiger«. Ich finde jedoch, dass es ein guter Stil ist.***

15.3.1 Anfügen am Kopf der verketteten Liste

Um zu sehen, wie verkettete Listen in der Praxis arbeiten, betrachten Sie die folgende einfache Beispielfunktion, die das ihr übergebene Argument an den Anfang der Liste anfügt:

```
void addHead(LinkableClass* pLC)
{
    pLC->pNext = pHead;
    pHead = pLC;
}
```

Dieser Prozess wird in Abbildung 15.2 grafisch dargestellt. Nach der ersten Zeile zeigt das Objekt `*pLC` auf das erste Objekt der Liste (das gleiche, auf das pHead zeigt), dargestellt als Schritt A. Nach der zweiten Anweisung zeigt der Kopfzeiger auf das übergebene Objekt `*pLC`, dargestellt als Schritt B.

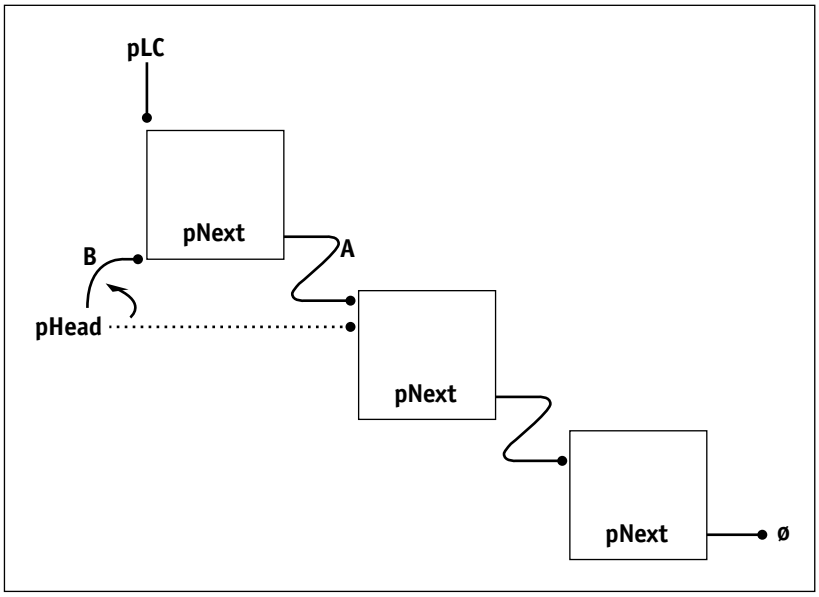


Abbildung 15.2: Ein Element wird in zwei Schritten am Kopf der Liste eingefügt.

15.3.2 Andere Operationen auf verketteten Listen

Ein Objekt am Kopf einer Liste einzufügen ist die einfachste der Operationen auf verketteten Listen. Ein Element am Ende der Liste einzufügen ist viel trickreicher.

```
void addTail(LinkableClass* pLC)
{
    // beginne mit einem Zeiger auf den Anfang
    // der verketteten Liste
    LinkableClass* pCurrent = pHead;

    // iteriere durch die Liste, bis wir das
    // letzte Element der Liste finden - das ist das
    // Element, dessen Zeiger pNext gleich null ist
    while(pCurrent->pNext != (LinkableClass*)0)
    {
        // bewege pCurrent zum nächsten Eintrag
        pCurrent = pCurrent->pNext;
    }

    // lasse das Objekt auf LC verweisen
    pCurrent->pNext = pLC;

    // stelle sicher, dass LC's pNext-Zeiger null
```

```
    // ist, wodurch LC als letztes Element in der
    // Liste markiert wird
    pLC->pNext = (LinkableClass*)0;
}
```

Die Funktion `addTail( )` beginnt mit einer Iteration, um das Element zu finden, dessen `pNext`-Zeiger gleich null ist – das ist das letzte Element in der Liste. Wenn dieses Element gefunden ist, verkettet `addTail( )` das Objekt `*pLC` mit dem Ende der Liste.

(Tatsächlich enthält die Funktion `addTail( )` so, wie wir sie geschrieben haben, einen Bug. Ein spezieller Test muss hinzugefügt werden, um festzustellen, ob `pHead` selbst null ist, was anzeigen würde, dass die Liste leer war).

Die Funktion `remove( )` ist ähnlich. Die Funktion entfernt das spezifizierte Objekt aus der Liste und gibt 1 zurück, wenn dies erfolgreich war, sonst 0.


```
int remove(LinkableClass* pLC)
{
    LinkableClass* pCurrent = pHead;

    // wenn die Liste leer ist, ist *pLC
    // offensichtlich nicht in der Liste
    if (pCurrent == (LinkableClass*)0)
    {
        return 0;
    }

    // iteriere durch die Schleife und suche
    // nach dem Element bis zum Ende der Liste
    while(pCurrent->pNext)
    {
        // wenn der nächste Eintrag *pLC ist ...
        if (pLC == pCurrent->pNext)
        {
            // ...dann soll der aktuelle Eintrag
            // auf dessen nächsten Eintrag verweisen
            pCurrent->pNext = pLC->pNext;

            // nicht absolut notwendig, aber entferne
            // das nächste Objekt aus *pLC, um nicht
            // verwirrt zu werden
            pLC->pNext = (LinkableClass*)0;
            return 1;
        }
    }
    return 0;
}
```

Die Funktion `remove( )` überprüft zuerst, ob die Liste auch nicht leer ist – wenn sie leer ist, gibt die Funktion den Fehlercode zurück, da offensichtlich das Objekt `*pLC` nicht in der Liste enthalten ist. Wenn die Liste nicht leer ist, iteriert `remove( )` durch alle Elemente, bis das Objekt gefunden wird, das auf `*pLC` verweist. Wenn dieses Objekt gefunden wird, setzt `remove( )` den Zeiger `pCurrent->pNext` an `*pLC` vorbei. Dieser Prozess wird in Abbildung 15.3 grafisch veranschaulicht.

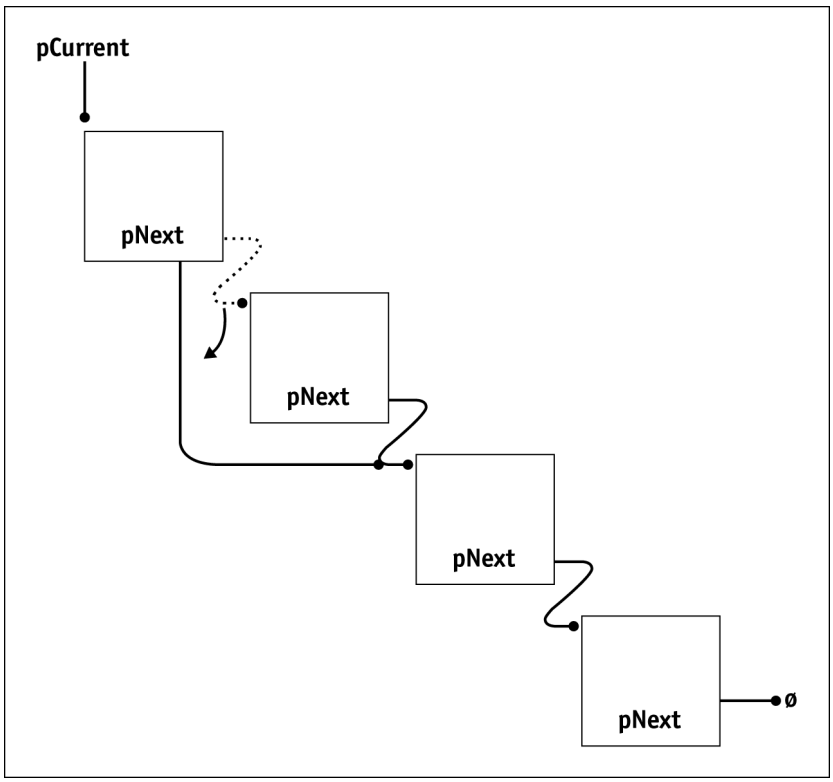
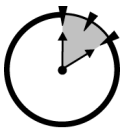


Abbildung 15.3: »Umgehe« einen Eintrag, um ihn aus der Liste zu entfernen

15.3.3 Eigenschaften verketteter Listen

Verkettete Listen sind all das, was Arrays nicht sind. Verkettete Listen können nach Belieben erweitert und verkleinert werden, indem Objekte eingefügt oder entfernt werden. Das Einfügen eines Objektes in die Mitte der verketteten Liste ist schnell und einfach – bereits eingefügte Elemente müssen nicht an eine andere Stelle kopiert werden. In gleicher Weise ist das Sortieren von Elementen in einer verketteten Liste schneller durchzuführen als in einem Array. Array-Elemente sind direkt über einen Index ansprechbar – eine damit vergleichbare Eigenschaft besitzen die verketteten Listen nicht. Programme müssen manchmal die gesamte Liste durchsuchen, um einen bestimmten Eintrag zu finden.



10 Min.

15.4 Ein Programm mit verkettetem NameData

Das Programm `LinkedListData`, das Sie im Folgenden finden, ist eine Version des Array-basierten Programms `ClassData` aus Sitzung 12, die eine verkette Liste verwendet.

```
// LinkedListData - speichere Namensdaten in einer
//                  verketteten Liste von Objekten
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet - speichere Vorname, Name und
//               Sozialversicherungsnummer
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
        int  nSocialSecurity;

        // der Verweis auf das nächste Listenelement
        NameDataSet* pNext;
};

// der Zeiger auf den ersten Eintrag der Liste
NameDataSet* pHead = 0;

// addTail - füge ein neues Element der Liste an
void addTail(NameDataSet* pNDS)
{
    // stelle sicher, dass der Zeigern von *pDNS null
    // ist, weil das zum letzten Listenelement wird
    pNDS->pNext = 0;

    // wenn die Liste leer ist, dann zeige einfach
    // mit dem Kopfzeiger auf den aktuellen Eintrag
    // und fertig
    if (pHead == 0)
    {
        pHead = pNDS;
        return;
    }

    // ansonsten finde das letzte Element der Liste
    NameDataSet* pCurrent = pHead;
    while(pCurrent->pNext)
    {
        pCurrent = pCurrent->pNext;
    }

    // jetzt füge den aktuellen Eintrag diesem an
    pCurrent->pNext = pNDS;
}

// getData - lies einen neuen Namen und eine
```

```
//          Sozialversicherungsnummer; gib null
//          zurück, wenn nichts mehr zu lesen ist
NameDataSet* getData()
{
    // neuer Eintrag, der zu füllen ist
    NameDataSet* pNDS = new NameDataSet;

    // lies den Vornamen
    cout << »\nVorname:«;
    cin  >> pNDS->szFirstName;

    // wenn der Vorname 'ende' oder 'ENDE' ist...
    if ((strcmp(pNDS->szFirstName, »ende«) == 0)
        ||
        (strcmp(pNDS->szFirstName, »ENDE«) == 0))
    {
        // ... lösche das immer noch leere Objekt ...
        delete pNDS;

        // ... gib eine null zurück für Eingabeende
        return 0;
    }

    // lies die übrigen Elemente
    cout << »Nachname:«;
    cin  >> pNDS->szLastName;

    cout << »Sozialversicherungsnummer:«;
    cin  >> pNDS->nSocialSecurity;

    // Zeiger auf nächstes Element auf null setzen
    pNDS->pNext = 0;

    // gib die Adresse auf das neue Element zurück
    return pNDS;
}

// displayData - Ausgabe des Datensatzes
//               auf den pDNS zeigt
void displayData(NameDataSet* pNDS)
{
    cout << pNDS->szFirstName
         << » »
         << pNDS->szLastName
         << »/«
         << pNDS->nSocialSecurity
         << »\n«;
}

int main(int nArg, char* pszArgs[])
{
    cout << »Lies Vornamen, Nachnamen und\n«
         << »Sozialversicherungsnummer\n«;
         << »Geben Sie 'ende' als Vorname ein, um\n«;
         << »das Programm zu beenden\n«;
```

```
    // erzeuge (weiteres) NameDataSet-Objekt
    NameDataSet* pNDS;
    while (pNDS = getData())
    {
        // füge es an die Liste der
        // NameDataSet-Objekte an
        addTail(pNDS);
    }

    // um die Objekte anzuzeigen, iteriere durch die
    // Liste (stoppe, wenn die nächste Adresse
    // null ist)
    cout << »Entries:\n«;
    pNDS = pHead;
    while(pNDS)
    {
        // Anzeige des aktuellen Eintrags
        displayData(pNDS);

        // gehe zum nächsten Eintrag
        pNDS = pNDS->pNext;
    }
    return 0;
}
```


Obwohl es in gewisser Hinsicht lang ist, ist das Programm `LinkedListData` doch recht einfach. Die Funktion `main( )` beginnt mit dem Aufruf von `getData( )`, um einen `NameDataSet`-Eintrag vom Benutzer zu bekommen. Wenn der Benutzer `ende` eingibt, gibt `getData( )` null zurück. `main( )` ruft dann `addTail( )` auf, um den Eintrag, der von `getData( )` zurückgegeben wurde, an das Ende der verketteten Liste anzufügen.

Sobald es vom Benutzer keine weiteren `NameDataSet-`Objekte gibt, iteriert `main( )` durch die Liste, und zeigt jedes Objekt mittels der Funktion `displayData( )` an.

Die Funktion `getData( )` alloziert zuerst ein leeres `NameDataSet-`Objekt von `Heap`. `getData( )` liest dann den Vornamen des einzufügenden Eintrags. Wenn der Benutzer als Vornamen `ende` oder `ENDE` eingibt, löscht die Funktion das Objekt, und gibt null an den Aufrufenden zurück. `getData( )` fährt mit dem Lesen des Nachnamens und der Sozialversicherungsnummer fort. Schließlich setzt `getData( )` den Zeiger `pNext` auf null, bevor die Funktion zurückkehrt.

Lassen Sie Zeiger niemals uninitialisiert. Wenden Sie die alte Programmiererregel an: »Im Zweifelsfalle ausnullen«.

Die Funktion `addTail( )`, die hier auftaucht, ist der Funktion `addTail( )` sehr ähnlich, die bereits in diesem Kapitel dargestellt wurde. Anders als die ältere Version, überprüft diese Version von `addTail( )`, ob die Liste leer ist, bevor sie startet. Wenn `pHead` null ist, dann setzt `addTail( )` den Zeiger `pHead` auf den aktuellen Eintrag und terminiert.

Die Funktion `displayData( )` ist eine Zeiger-basiere Version der früheren Funktionen `display-Data( )`.

0 Min.

15.5 Andere Container

Ein *Container* ist eine Struktur, die entworfen wurde, um Objekte zu enthalten. Arrays und verkettete Listen sind spezielle Container. Der Heap ist auch eine Form Container; er enthält einen separaten Speicherblock, der dem Programm zur Verfügung steht.

Sie haben möglicherweise von anderen Containern gehört, wie z.B. der FIFO (first-in-first-out) und der LIFO (last-in-first-out); der Container LIFO wird auch als Stack (Stapel) bezeichnet. Diese stellen zwei Funktionen bereit, jeweils eine für das Einfügen und das Löschen von Objekten. Die FIFO entfernt das älteste Objekt, während die LIFO das jüngste Objekt entfernt.

Zusammenfassung

Zeiger auf Klassenobjekte machen es möglich, den Wert von Klassenobjekten innerhalb von Funktionen zu verändern. Eine Referenz auf ein Klassenobjekt zu übergeben, ist bedeutend schneller, als ein Klassenobjekt als Wert zu übergeben. Ein Zeigerelement in eine Klasse einzufügen, ermöglicht eine Verkettung der Objekte in einer verketteten Liste. Die Struktur der verketteten Liste bietet mehrere Vorteile gegenüber Arrays, während andererseits Effizienz eingebüßt wird.

- Zeiger auf Klassenobjekte arbeiten im Wesentlichen wie Zeiger auf andere Datentypen. Dies umfasst die Fähigkeit, Objekte als Referenz an eine Funktion zu übergeben.
- Ein Zeiger auf ein lokales Objekt ist nicht mehr gültig, wenn die Kontrolle von der Funktion zurückgegeben wurde. Objekte, die vom Heap alloziert wurden, haben keine solche Beschränkung ihres Gültigkeitsbereiches und können daher von Funktion zu Funktion übergeben werden. Es obliegt jedoch dem Programmierer, Objekte, die vom Heap alloziert wurden, an den Heap zurückzugeben; geschieht dies nicht, sind fatale und schwer zu findende Speicherlöcher die Folge.
- Objekte können in einer verketteten Liste miteinander verbunden werden, wenn ihre Klasse einen Zeiger auf ein Objekt ihres eigenen Typs enthält. Es ist einfach, Elemente in die verkettete Liste einzufügen, und Elemente aus der verketteten Liste zu löschen. Obwohl wir das hier nicht gezeigt haben, ist das Sortieren einer verketteten Liste einfacher als das Sortieren eines Array. Objektzeiger sind auch nützlich bei der Erzeugung anderer Container, die hier nicht vorgestellt wurden.

Selbsttest

1. Gegeben sei die folgende Klasse:

```
class MyClass
{
    int n;
}
MyClass* pM;
```

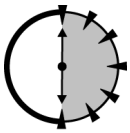
Wie würden Sie das Datenelement `n` vom Zeiger `pM` referenzieren? (Siehe »Zeiger auf Objekte«)

2. Was ist ein Container? (Siehe »Andere Container«)
3. Was ist eine verkettete Liste? (Siehe »Verkettete Listen«)
4. Was ist ein Kopfzeiger? (Siehe »Verkettete Listen«)

Debuggen II

Checkliste

- ☑ Schrittweise durch ein Programm
- ☑ Haltepunkte setzen
- ☑ Variablen ansehen und modifizieren
- ☑ Ein Programm mit einem Debugger debuggen



30 Min.

Sitzung 10 hat eine Technik zum Debuggen von Programmen vorgestellt, die auf der Ausgabe von Schlüsselinformationen auf `cout` basiert. Wir haben diese so genannte Technik der Ausgabeanweisungen verwendet, um das zugegebenermaßen einfache Beispielprogramm ErrorProgram zu debuggen.

Für kleinere Programme arbeitet diese Technik gut. Probleme mit dieser Technik treten erst auf, wenn der Umfang der Programme über den der hier dargestellten Beispielprogramme hinausgeht.

In größeren Programmen weiß der Programmierer oft nicht, wo er Ausgabeanweisungen hinsetzen muss. Ein strenger Zyklus von Einfügen von Ausgabeanweisungen, Erzeugen des Programms, Ausführen des Programms, Einfügen von Ausgabeanweisungen usw. ist nervig. Um Ausgabeanweisungen zu verändern, muss das Programm stets neu erzeugt werden. Bei einem großen Programm kann selbst die Zeit für die Erzeugung schon beachtlich sein.

Ein zweiter, ausgeklügelterer Ansatz basiert auf einem separaten Werkzeug, dem Debugger. Dieser Zugang vermeidet viele der Nachteile, die in der Technik der Ausgabeanweisungen enthalten sind. Diese Sitzung führt Sie in die Verwendung des Debuggers ein, indem wir den Bug in einem kleinen Programm finden.

Ein großer Teil dieses Buches ist dem Studium der Programmierfähigkeiten gewidmet, die durch Zeigervariablen ermöglicht werden. Zeiger haben jedoch auch ihren Preis: Zeigerfehler sind leicht zu begehen, und extrem schwierig zu finden. Die Technik der Ausgabeanweisungen taugt für das Finden und Entfernen von Zeigerfehlern nicht. Nur ein guter Debugger kann bei solchen Fehlern helfen.

16.1 Welcher Debugger?

Anders als bei der Programmiersprache C++, die über Herstellergrenzen hinweg standardisiert ist, hat jeder Debugger seine eigenen Kommandos. Glücklicherweise bieten die meisten Debugger die gleichen elementaren Kommandos. Die Kommandos, die wir benötigen, sind sowohl in Visual C++ als auch in der `rhide`-Umgebung von GNU C++ enthalten. Beide Umgebungen bieten diese Basiskommandos über Pulldown-Menüs an. Beide Debugger bieten auch den schnellen Zugriff auf die wichtigsten Debuggerfunktionen über Tastenkombinationen. Tabelle 16-1 listet diese Kommandos für beide Umgebungen auf.

Im Rest dieser Sitzung werden ich Debugkommandos über ihren Namen ansprechen. Verwenden Sie Tabelle 16-1 um die entsprechende Tastenkombination zu finden.

Tabelle 16-1: Debuggerkommandos für Visual C++ und GNU C++

Kommando	Visual C++	GNU C++ (rhide)
Build	Shift+F8	F9
Step In	F11	F7
Step Over	F10	F8
View Variable	siehe Text	Ctl+F4
Set Breakpoint	F9	Ctl+F8
Add Watch	siehe Text	Ctl+F7
Go	F5	Ctl+F9
View User Screen	Klicken auf Programmfenster	Alt+F5
Program Reset	Shift+F5	Ctl+F2

Um Verwirrung hinsichtlich der kleinen Unterschiede der beiden Debugger zu vermeiden, beschreibe ich den Debugprozess, den ich mit `rhide` verfolgt habe. Danach debugge ich das gleiche Programm noch einmal mit dem Debugger von Visual C++.

16.2 Das Testprogramm

Ich habe das folgende Programm geschrieben, das einen Bug (Fehler) enthält. Das Schreiben fehlerhafter Programme ist für mich nicht schwierig, weil meine Programme fast nie beim ersten Mal laufen.

Die Datei ist auf der beiliegenden CD-ROM unter dem Namen Concatenate(Error).cpp zu finden.

```
// Concatenate - verbinde zwei Zeichenketten
//               mit » - « in der Mitte
//               (diese Version stürzt ab)
#include <stdio.h>
#include <iostream.h>
void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    cout << »Dieses Programm verbindet zwei Zeichenketten\n«;
    cout << »(Diese Version stürzt ab)\n\n«;

    // lese erste Zeichenkette ...
    char szString1[256];
    cout << »Zeichenkette #1:«;
    cin.getline(szString1, 128);

    // ... nun die zweite Zeichenkette ...
    char szString2[128];
    cout << »Zeichenkette #2:«;
    cin.getline(szString2, 128);

    // ... füge » - « an die erste an ...
    concatString(szString1, » - «);

    // ... nun füge zweite an ...
    concatString(szString1, szString2);

    // ... und zeige das Resultat
    cout << »\n« << szString1 << »\n«;

    return 0;
}

// concatString - fügt die Zeichenkette szSource
//                an das Ende von szTarget an
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex;
    int nSourceIndex;

    // finde das Ende der ersten Zeichenkette
    while(szTarget[++nTargetIndex])
    {
    }

    // füge die zweite ans Ende der ersten an
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }
}
```

Das Programm kann ohne Fehler erzeugt werden. Ich führe das Programm aus. Das Programm fragt nach Zeichenkette #1, und ich gebe »das ist eine Zeichenkette« ein. Für Zeichenkette #2 gebe ich DAS IST EINE ZEICHENKETTE ein. Aber anstatt die korrekte Ausgabe zu erzeugen, bricht das Programm mit Fehlercode 0xff ab. Ich drücke OK. Der Debugger versucht, mir ein wenig Trost zu spenden, indem er das Fenster unterhalb des Eingabefensters öffnet, wie in Abbildung 16.1 zu sehen ist.

Die erste Zeile im Nachrichtenfenster zeigt an, dass `rhide` denkt, dass der Fehler in Zeile 46 des Moduls `Concatenate(error1)` aufgetreten ist. Außerdem wurde die Funktion, die abgestürzt ist, von Zeile 29 im gleichen Modul aufgerufen. Das weist scheinbar darauf hin, dass die initiale `while`-Schleife innerhalb von `concatString( )` fehlerhaft ist.

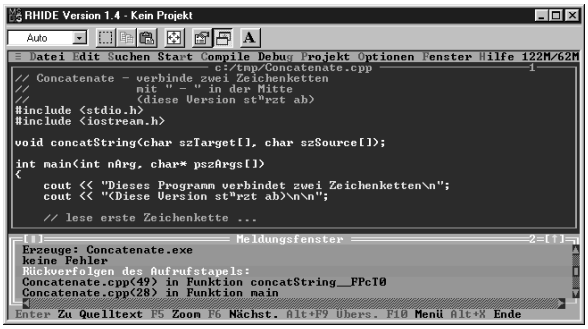
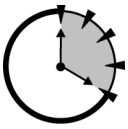


Abbildung 16.1: Der rhide-Debugger gibt einen Hinweis auf die Fehlerquelle, wenn ein Programm abstürzt.

Weil ich in dieser Anweisung keinen Fehler finden kann, mache ich den Debugger zu meinem Gehilfen.

Tatsächlich sehe ich das Problem, ausgehend von den Informationen, die `rhide` ***mir liefert. Doch gehen Sie weiter mit mir durch.***



20 Min.

16.3 Einzelschritte durch ein Programm

Ich drücke Step Over, um mit dem Debuggen des Programms zu beginnen. `rhide` öffnet ein MS-DOS-Fenster, als wenn es das Programm ausführen wollte. Doch bevor das Programm mit der Ausführung beginnen kann, schließt der Debugger das Programmfenster und zeigt das Editierfenster des Programms an, in dem die erste ausführbare Zeile des Programms markiert ist.

Eine *ausführbare Anweisung* ist eine Anweisung, die keine Deklaration oder Kommentar ist. Eine ausführbare Anweisung ist eine Anweisung, die beim Kompilieren Maschinencode erzeugt.

Der Debugger hat tatsächlich das Programm ausgeführt bis zur ersten Zeile der Funktion `main( )` und dann dem Programm die Kontrolle entzogen. Der Debugger wartet auf Sie, zu entscheiden, wie es weitergehen soll.

Indem ich mehrfach Step Over drücke, kann ich das Programm ausführen, bis es zum Absturz kommt. Das sagt mir viel darüber, was falsch gelaufen ist.

Ein Programm zeilenweise auszuführen wird auch als *Einzelschrittausführung* eines Programms bezeichnet.

Als ich Step Over für das Kommando `cin.getline( )` ausführe, bekommt der Debugger die Kontrolle vom MS-DOS-Fenster nicht wie üblich zurück. Stattdessen scheint das Programm beim Prompt eingefroren zu sein und darauf zu warten, dass die erste Zeichenkette eingegeben wird.

In der Nachbetrachtung merke ich, dass der Debugger die Kontrolle vom Programm erst dann zuruckbekommt, wenn die C++-Anweisung ausgeführt ist – eine Anweisung, die den Aufruf `getline( )` enthält, ist erst dann ausgeführt, wenn ich einen Text über die Tastatur eingegeben habe.

Ich gebe eine Zeile Text ein und drücke die Entertaste. Der `rhide`-Debugger hält das Programm bei der nächsten Anweisung an: `cout << »Zeichenkette #2:«`. Wieder führe ich einen Einzelschritt aus, indem ich die zweite Zeile Text eingebe als Antwort auf den zweiten Aufruf von `getline( )`.

Wenn der Debugger anzuhalten scheint, ohne zurückzukommen, wenn Sie in Einzelschritten durch ein Programm gehen, wartet Ihr Programm darauf, dass etwas Bestimmtes passiert. Am wahrscheinlichsten ist, dass das Programm auf eine Eingabe wartet, entweder von Ihnen oder von einem externen Device.

Schließlich gehe ich im Einzelschrittmodus durch den Aufruf von `concatString( )`, wie in Abbildung 16.2 zu sehen ist. Als ich Step Over für den Aufruf versuche, stürzt das Programm wie zuvor ab.

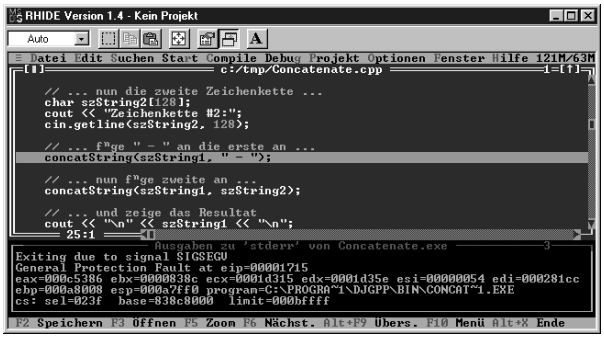


Abbildung 16.2: Etwas in der Funktion `concatString( )` ***verursacht den Absturz.***

Das sagt mit nicht mehr als ich schon vorher wusste. Was ich brauche, ist die Möglichkeit, Einzelschritte in der Funktion auszuführen, statt einen Schritt über die Funktion hinweg zu machen.

16.4 Einzelschritte in eine Funktion hinein

Ich entscheide mich für einen weiteren Versuch. Erst drücke ich Reset, um den Debugger am Anfang des Programms zu starten.

Denken Sie immer daran, Program Reset zu drücken, bevor Sie erneut beginnen. Es tut nicht weh, den Knopf oft zu drücken. Sie können sich daran gewöhnen, Program Reset jedes Mal auszuführen, bevor Sie den Debugger starten.

Wieder gehe ich in Einzelschritten durch das Programm unter Verwendung von Step Over, bis ich den Aufruf von `concatString( )` erreiche. Diesmal rufe ich nicht Step Over, sondern Step In auf, um in die Funktion zu gelangen. Sofort bewegt sich die Markierung zur ersten ausführbaren Zeile innerhalb von `concatString( )` wie in Abbildung 16.3 zu sehen ist.

Es gibt keinen Unterschied zwischen Step Over und Step In, wenn kein Funktionsaufruf ausgeführt wird.

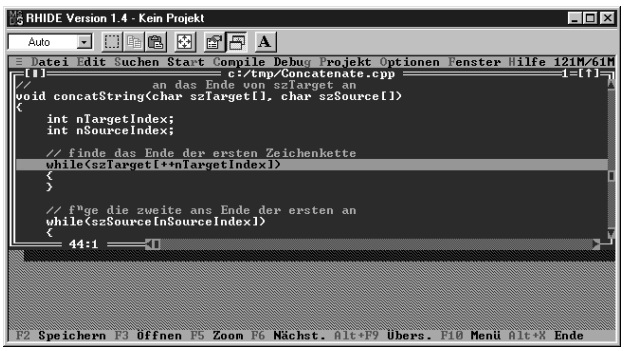


Abbildung 16.3: Das Kommando Step In bewegt die Kontrolle zur ersten ausführbaren Zeile von `concatString( )`.

Wenn Sie Step In versehentlich für eine Funktion ausgeführt haben, kann es sein, dass der Debugger Sie nach der Quelldatei einer Datei fragen wird, von der Sie vorher noch nie gehört haben. Das ist die Datei des Bibliothekmoduls, das die Funktion enthält, in die Sie hineingehen wollten. Drücken Sie Cancel, und Sie erhalten eine Liste von Maschineninstruktion, die selbst für die härtesten Techniker nicht sehr hilfreich ist. Um wieder in einen gesunden Zustand zu kommen, öffnen Sie das Eingabefenster, setzen einen Haltepunkt, wie im nächsten Abschnitt beschrieben, auf die Anweisung direkt nach dem Aufruf und drücken Go.

Mit großer Hoffnung drücke ich Step Over, um die erste Anweisung in der Funktion auszuführen. Der `rhide`-Debugger antwortet mit der Meldung eines Segmentfehlers wie in Abbildung 16.4 zu sehen ist.

Abbildung 16.4: Ein Einzelschritt auf der ersten Zeile von `concatString( )` ***erzeugt einen Segmentfehler.***

Ein Segmentfehler zeigt im Allgemeinen an, dass ein Programm auf ein ungültiges Speichersegment zugegriffen hat, entweder weil ein Zeiger ungültig geworden ist, oder ein Array außerhalb seiner Grenzen adressiert wurde. Um es interessanter zu machen, lassen Sie uns annehmen, dass ich das nicht weiß.

Jetzt weiß ich sicher, das etwas in der `while`-Schleife nicht korrekt ist, und dass bereits die erste Ausführung der Schleife zum Absturz führt. Um herauszufinden, was schiefgeht, muss ich das Programm unmittelbar vor der fehlerhaften Zeile anhalten.

16.5 Verwendung von Haltepunkten

Wieder drücke ich Program Reset, um den Debugger auf den Anfang des Programms zurückzubringen. Ich könnte wieder in Einzelschritten durch das Programm gehen, bis ich auf die `while`-Schleife treffe. Stattdessen wähle ich eine Abkürzung. Ich platziere den Cursor auf der `while`-Anweisung und führe das Kommando Set Breakpoint aus. Der Editor markiert die Anweisung rot, wie in Abbildung 16.5 zu sehen ist.

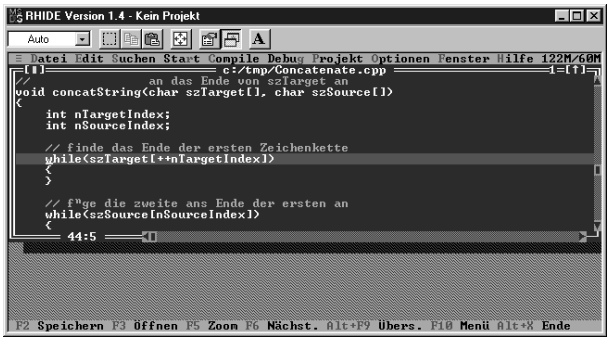
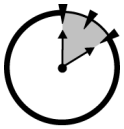


Abbildung 16.5: Das Kommando Set Breakpoint.

Ein *Haltepunkt* teilt dem Debugger mit, bei dieser Anweisung anzuhalten, wenn die Kontrolle jemals dorthin gelangt. Ein Haltepunkt lässt ein Programm wie gewohnt ablaufen bis zu diesem Punkt, an dem wir die Kontrolle übernehmen möchten. Haltepunkte sind nützlich, wenn wir wissen, wo wir anhalten möchten, oder wenn wir das Programm normal ausführen möchten, bis es Zeit zum Anhalten ist.

Nachdem ich den Haltepunkt gesetzt habe, drücke ich Go. Das Programm scheint normal ausgeführt zu werden bis zum Aufruf von `while`. An diesem Punkt springt das Programm zurück zum Debugger.



10 Min.

16.6 Ansehen und Modifizieren von Variablen

Es macht nicht viel Sinn, die `while`-Anweisung wieder auszuführen – ich weiß ja, dass sie zum Absturz führen wird. Ich brauche mehr Informationen darüber, was das Programm macht, um festzustellen, was zum Absturz führt. Z.B. möchte ich gerne den Inhalt der Variablen `nTargetIndex` unmittelbar vor der Ausführung der `while`-Schleife sehen.

Zuerst führe ich einen Doppelklick auf dem Variablennamen `nTargetIndex` aus. Dann drücke ich View Variable. Ein Fenster erscheint, mit dem Namen nTargetIndex im oberen Feld. Ich drücke Eval, um den aktuellen Wert der Variable herauszufinden. Das Ergebnis, das Sie in Abbildung 1.6 finden, macht offensichtlich keinen Sinn.

Bei einem Blick zurück in den C++-Code stelle ich fest, dass ich die Variablen `nTargetIndex` und `nSourceIndex` nicht initialisiert habe. Um das zu überprüfen, gebe ich 0 im Fenster New Value ein und drücke Change. Ich führe das Gleiche für `nSourceIndex` aus. Ich schließe das Fenster und drücke Step Over, um die Ausführung fortzusetzen.

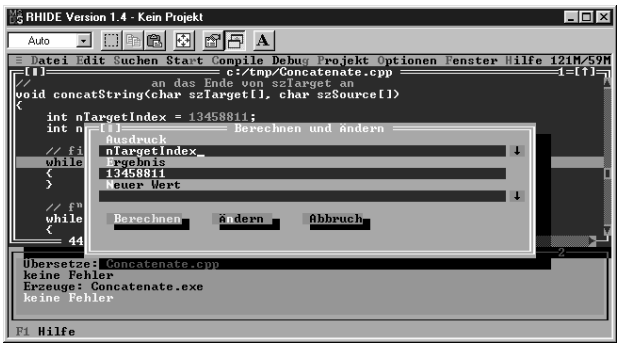


Abbildung 16.6: Dieses Fenster erlaubt es dem Programmierer, gültige Variablen anzusehen und ihren Wert zu verändern.

Mit den nun initialisierten Indexvariablen gehe ich in Einzelschritten durch die `while`-Schleife hindurch. Jeder Aufruf von Step Over oder Step In führt eine Iteration der `while`-Schleife aus. Weil der Cursor nach dem Aufruf da steht, wo er vorher auch war, tritt scheinbar keine Veränderung auf; nach einem Schleifendurchlauf hat `nTargetIndex` jedoch den Wert 1.

Weil ich mir nicht die Arbeit machen möchte, den Wert von `nTargetIndex` nach jeder Interation zu überprüfen, führe ich einen Doppelklick auf `nTargetIndex` aus, und führe das Kommando Add Watch aus. Es erscheint ein Fenster mit der Variable nTargetIndex und dem Wert 1 rechts daneben. Ich drücke mehrmals Step In, und in jeder Iteration wird der Wert von `nTargetIndex` um eins erhöht. Nach einigen Iterationen wird die Kontrolle an eine Anweisung außerhalb der Schleife übergeben.

Ich setze einen Haltepunkt auf die schließende Klammer der Funktion `concatString( )` und drücke Go. Das Programm hält unmittelbar vor der Rückkehr der Funktion an.

Um die erzeugte Zeichenkette zu überprüfen, führe ich eine Doppelklick auf `szTarget` aus und drücke View Variable. Die Ergebnisse, die in Abbildung 16.7 zu sehen sind, sind nicht so, wie ich es erwartet habe.

Die 0x6ccee0 ist die Adresse der Zeichenkette im Speicher. Diese Information kann hilfreich sein, wenn es um Zeiger geht. Diese Information könnte z.B. sehr hilfreich sein beim Debuggen einer Anwendung, die verkettete Listen verwendet.

Es sieht so aus, als ob die Zielzeichenkette nicht verändert worden wäre, obwohl ich genau weiß, dass die zweite `while`-Schleife ausgeführt wurde. Mit einer kleinen Chance, dass die zweite Zeichenkette doch da ist, sehe ich hinter der initialen Zeichenkette nach. `szTarget + 27` sollte die Adresse des ersten Zeichens nach der `Null` der Zeichenkette »DIES IST EINE ZEICHENKETTE«, die ich eingegeben habe, sein. Und tatsächlich, das » – « steht da, gefolgt von einem »d«, das korrekt zu sein scheint. Das ist in Abbildung 16.8 zu sehen.

Abbildung 16.7: Die Zielzeichenkette scheint nach Rückkehr der Funktion `concatString( )` ***nicht verändert zu sein.***

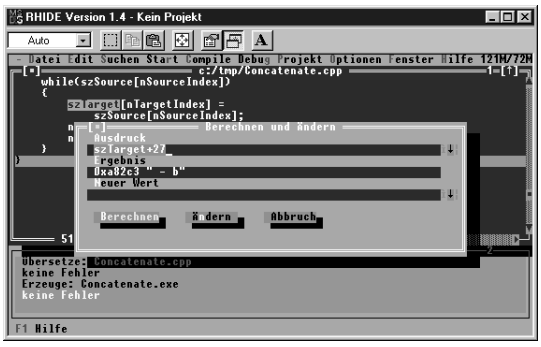


Abbildung 16.8: Die Quellzeichenkette scheint an einer falschen Stelle an die Zielzeichenkette angefügt worden zu sein.

Nach reiflicher Überlegung ist es offensichtlich, dass `szSource` hinter dem abschließenden `Null`-zeichen an `szTarget` angefügt wurde. Zusätzlich ist klar, dass die resultierende Ergebniszeichenkette überhaupt nicht abgeschlossen wurde (daher das »D« am Ende).

Eine Zeichenkette hinter dem `Nullzeichen` zu verändern, oder das terminierende `Nullzeichen` zu vergessen, sind die häufigsten Fehler beim Umgang mit Zeichenketten.

Weil ich jetzt zwei Fehler kenne, drücke ich Program Reset und berichtige die Funktion `concatString( )`, so dass die zweite Zeichenkette an der richtigen Stelle eingefügt wird und die Ergebniszeichenkette mit einem Nullzeichen abgeschlossen wird. Die geänderte Funktion `concatString( )` sieht wie folgt aus:

```
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex = 0;
    int nSourceIndex = 0;

    // finde das Ende der ersten Zeichenkette
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // füge die zweite ans Ende der ersten an
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }

    // terminiere die Zeichenkette
    szTarget[nTargetIndex] = '\0';
}
```

Weil ich vermute, dass es noch ein weiteres Problem gibt, beobachte ich `szTarget` und `nTarget-Index`, während die zweite Schleife ausgeführt wird. Nun wird die Zeichenkette korrekt ans Ende der Zielzeichenkette kopiert, wie in Abbildung 16.9 zu sehen ist. (Abbildung 16.9 zeigt den zweiten Aufruf von `concatString( )`, weil er besser zu verstehen ist.)

Sie müssen das wirklich selber ausführen. Das ist der einzige Weg, wie Sie ein Gefühl dafür bekommen könne, wie hübsch es ist, einer Zeichenkette beim Wachsen zuzusehen, während die andere Zeichenkette in jeder Interation der Schleife schrumpft.

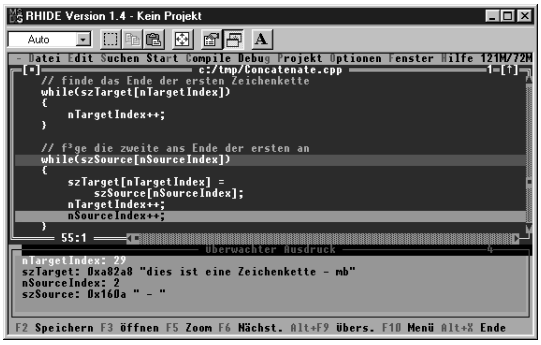


Abbildung 16.9: Zeigt wie die Quellzeichenkette an das Ende der Zielzeichenkette gehängt wird.

Bei der erneuten Untersuchung der Zeichenkette unmittelbar vor dem Anfügen der terminierenden `Null`, stelle ich fest, dass die Zeichenkette `szTarget` korrekt ist mit Ausnahme des Extrazeichens am Ende, wie in Abbildung 16.10 zu sehen ist.

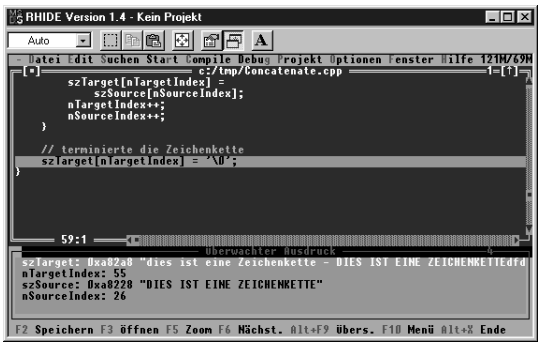


Abbildung 16.10: Vor dem Hinzufügen des Nullzeichens enthält die Ergebniszeichenkette am Ende weitere Zeichen.

Sobald ich Step Over drücke, fügt das Programm das Nullzeichen an und die »Rauschzeichen« verschwinden aus dem Fenster.

16.7 Verwendung des Visual C++ Debuggers

Die Schritte, um das Programm `Concatenate` mit den Debugger von Visual C++ zu debuggen, sind ähnlich zu den Schritten, die wir mit `rhide` ausgeführt haben. Der Hauptunterschied ist jedoch, dass der Debugger von Visual C++ das auszuführende Programm in einem separaten MS-DOS-Fenster öffnet, statt als Teil des Debuggers selber. Wenn Sie Go drücken, erscheint in der Programmleiste von Windows ein neues Icon, das den Namen des Programms zeigt, in diesem Fall Concatenate. Der Programmierer kann das Benutzerfenster ansehen, indem er das Programmfenster auswählt.

Ein zweiter Unterschied ist die Art und Weise, mit der der Visual C++-Debugger das Ansehen von Variablen löst. Wenn die Ausführung an einem Haltepunkt gestoppt ist, kann der Programmierer einfach den Cursor über eine Variable bewegen. Wenn die Variable im Gültigkeitsbereich ist, zeigt der Debugger ihren Wert in einem kleinen Fenster, wie in Abbildung 16.11 zu sehen ist.

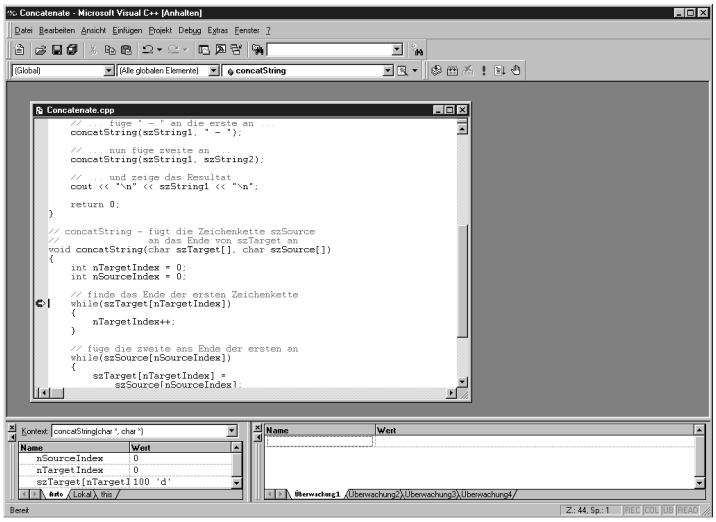


Abbildung 16.11: Visual C++ zeigt den Wert einer Variable, wenn der Cursor über sie bewegt wird.

0 Min.

Zusätzlich bietet der Debugger von Visual C++ einen bequemen Weg, um lokale Variablen (das sind Variable, die lokal in einer Funktion deklariert sind) anzusehen. Wählen Sie View, dann Debug Windows und schließlich Variables. Vom Fenster Variables aus wählen Sie die Locals-Schaltfäche. Alternativ können Sie Alt+4 drücken. Dieses Fenster markiert sogar die Variablen, die seit dem letzen Haltepunkt verändert wurden. Abbildung 16.12 zeigt dieses Fenster, während Einzelschritte durch das Kopieren der Quelle in die Zielzeichenkette ausgeführt werden.

Das Zeichen »I« am Ende von `szTarget` ***spiegelt die Tatsache wieder, dass die Zeichenkette noch nicht terminiert wurde.***

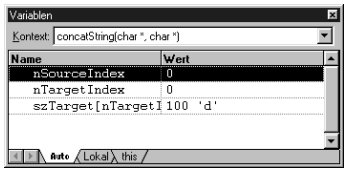


Abbildung 16.12: Dieses Fenster des Visual C++-Debuggers zeigt den Wert von Variablen, während Einzelschritte ausgeführt werden.

Zusammenfassung

Lassen Sie uns den Debugger-Zugang zum Finden eines Problems vergleichen mit dem Zugang über Ausgabeanweisungen, den wir in Sitzung 10 eingeführt haben. Der Debugger-Zugang ist nicht einfach zu erlernen. Ich bin mir sicher, dass Ihnen viele der hier eingegebenen Kommandos fremd vorgekommen sind. Wenn Sie sich jedoch erst einmal an den Debugger gewöhnt haben, können Sie ihn nutzen, um viel über Ihr Programm zu erfahren. Die Fähigkeit, langsam durch Ihr Programm durchzugehen, während Sie Variablen ansehen und verändern können, ist ein mächtiges Werkzeug.

Ich bevorzuge es, den Debugger beim ersten Aufruf eines neuen Programms aufzurufen. Schrittweise durch ein Programm durchzugehen, schafft ein gutes Verständnis dafür, was wirklich ausgeführt wird.

Ich war gezwungen, das Programm mehrmals auszuführen, als ich den Debugger verwendet habe. Das Programm musste ich jedoch nur einmal kompilieren und erzeugen, obwohl ich mehr als einen Fehler gefunden habe. Das ist ein großer Vorteil, wenn Sie ein großes Programm debuggen, das einige Minuten für seine Erzeugung benötigt.

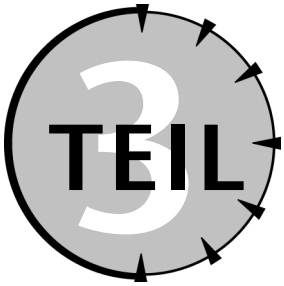
Ich habe schon an Projekten mitgearbeitet, bei denen der Computer die ganze Nacht damit beschäftigt war, das System neu zu erzeugen. Während das eher die Ausnahme ist, sind 5 bis 30 Minuten für die Erzeugung realer Applikationen nicht außergewöhnlich.

Schließlich gibt der Debugger Ihnen Zugriff auf Informationen, die Sie nicht so einfach sehen könnten, wenn Sie den Zugang der Ausgabeanweisungen wählen. Z.B. ist das Ansehen eines Zeigerinhaltes einfach mit einem Debugger. Es ist zwar möglich, doch sehr umständlich, immer wieder Adresseninformationen auszugeben.

Selbsttest

1. Was ist der Unterschied zwischen Step Over und Step In? (Siehe »Einzelschritte in eine Funktion hinein«)
2. Was ist ein Haltepunkt? (Siehe »Verwendung von Haltepunkten«)
3. Was ist eine Watch? (Siehe »Ansehen und Modifizieren von Variablen«)

Samstagnachmittag – Zusammenfassung



1. *Definieren Sie eine Studentenklasse, die den Nachnamen des Studenten, den Grad (erster Grad, zweiter Grad, usw.) und den mittleren Grad speichern kann.*
2. *Schreiben Sie eine Funktion, um ein Studentenobjekt zu lesen und seine Informationen auszugeben.*
3. *Geben Sie drei Grade ein, und mitteln Sie diese, bevor Sie das Objekt wieder ausgeben.*

   Hinweise:

   a. Verwenden Sie eine Abbildung von Zahlen auf Grade. Z.B. sei 1 der erste Grad, 2 der zweite Grad, usw.

   b. Der mittlere Grad ist eine Gleitkommazahl.
4. *Wenn die folgenden Variablen im Speicher ohne Zwischenräume angeordnet sind, wie viele Bytes Speicher beansprucht jede Variable in einem Programm, das mit Visual C++ oder GNU C++ erzeugt wurde:*

   a. `int n1; long l1; double d1;`

   b. `int nArray[20];`

   c. `int* pnPt1; double* pdPt2;`
5. *Betrachten Sie die folgende Funktion:*

```
void fn(int n1)
{
    int* pnVar1 = new int;
    *pnVar1 = n1;
}
```

   a. Kann die Funktion kompiliert werden?

   b. Was ist falsch an ihr?

   c. Warum könnte die Funktion Probleme verursachen?

   d. Was macht diese Art Problem so schwer auffindbar?

6. ***Beschreiben Sie die Speicheranordnung von*** `double dArray[3]`. ***Nehmen Sie an, dass das Array bei Adresse 0x100 beginnt.***

7. ***Unter Verwendung des Arrays in Aufgabe 6, beschreiben Sie den Effekt des Folgenden:***

```
double dArray[3];
double* pdPtr = &dArray[1];
*pdPtr = 1.0; // Zuweisung #1
int* pnPtr = (int*)&dArray[2];
*pnPtr = 2;   // Zuweisung #2
```

8. ***Schreiben Sie eine Funktion*** `LinkableClass* removeHead( )`, ***die das erste Element einer Liste von*** `LinkableClass`-***Objekten entfernt, und an den Aufrufenden zurückgibt.***

   Hinweise:

   a. Vergessen Sie nicht, das die Liste bereits leer sein könnte.

   b. Geben Sie null zurück, wenn die Liste leer ist.

   c. Wen Sie Schwierigkeiten mit leeren Listen haben, fangen Sie damit an, dass Sie die Liste als nicht leer annehmen. Nachdem Ihre Funktion fertig ist, versuchen Sie den Spezialfall einzubauen, dass die Liste leer ist.

9. ***Schreiben Sie eine Funktion*** `LinkableClass* returnPrevious(LinkableClass* pTarget)`, ***die den Vorgänger von*** `pTarget` ***in einer verketteten Liste zurückgibt, d.h. den Eintrag in der Liste, der auf*** `pTarget` ***verweist. Geben Sie null zurück, wenn die Liste leer ist, oder*** `pTarget` ***nicht gefunden wurde. Denken Sie wieder daran, auf das Ende der Liste zu achten.***

10. ***Schreiben Sie eine Funktion*** `LinkableClass* returnTail( )`, ***die den letzten Eintrag entfernt und an den Aufrufenden zurückgibt.***

    Hinweis: Erinnern Sie sich daran, dass der pNext-Zeiger des letzten Elementes gleich `null` ***ist.***

    Zusatzaufgabe: Schreiben Sie eine Funktion `LinkableClass* removeTail( )`, ***die das letzte Element der Liste entfernt, und dieses Element zurückliefert.***

    Hinweise:

    a. Versuchen Sie, die Funktion `returnPrevious( )` ***zu verwenden. Sie sollte in der Lage sein, die meiste Arbeit zu erledigen.***

    b. Wenn der Vorgängereintrag des letzen Elements null ist, dann hat die Liste nur einen Eintrag.

11. Updaten Sie das Programm Concatenate mit der folgenden Zeigerversion von `concatString`***, nachdem Sie mittels eines Debuggers (GNU C++ oder Visual C++) den darin enthaltenen Fehler entfernt haben:***

```
void concatString(char* pszTarget, char* pszSource)
{
    // hänge die zweite ans Ende der ersten
       while(*pszTarget)
    {
        *pszTarget++ = *pszSource++;
    }

    // terminiere die Zeichenkette
    *pszTarget = '\0';
}
```

Samstagabend

Teil 4

Lektion 17
Objektprogrammierung

Lektion 18
Aktive Klassen

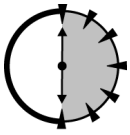
Lektion 19
Erhalten der Klassenintegrität

Lektion 20
Klassenkonstruktoren II

Objekt-programmierung

Checkliste

- ☑ Objekte in der realen Welt identifizieren
- ☑ Objekte in Klassen klassifizieren
- ☑ Objektorientierter und funktionaler Ansatz im Vergleich
- ☑ Nachos herstellen



30 Min.

Beispiele für Objektprogrammierung können im täglichen Leben gefunden werden. In der Tat sind Objekte überall. Direkt vor mir steht ein Stuhl, ein Tisch, ein Computer und ein halbgegessenes Brötchen. Objektprogrammierung wendet diese Konzepte auf die Welt des Programmierens an.

17.1 Abstraktion und Mikrowellen

Manchmal, wenn mein Sohn und ich Fußball schauen, bereite ich mir Unmengen von Nachos zu, die ich für fünf Minuten in die Mikrowelle stelle. (Nachos sind eine Spezialität mit Chips, Bohnen, Käse und Jalapenos).

Um die Mikrowelle zu benutzen, öffne ich die Tür, stelle die Sachen rein und drücke ein paar Knöpfe. Nach ein paar Minuten sind die Nachos fertig.

Das klingt nicht sehr profund, aber denken Sie einige Minuten über alle die Dinge nach, die ich nicht tue, um die Mikrowelle zu benutzen:

- Ich schaue nicht in das Innere der Mikrowelle; ich schaue nicht das Listing des Codes an, den der zentrale Prozessor ausführt; ich studiere auch nicht den Schaltplan des Gerätes.
- Ich ändere nichts an der Mikrowelle, um sie zum Laufen zu bringen, auch nichts an ihrer Verkabelung. Die Mikrowelle hat ein Interface, das mich alles ausführen lässt, was ich brauche – die Frontplatte mit all den Knöpfen und der kleinen Zeitanzeige.
- Ich schreibe nicht das Programm neu, das auf dem kleinen Prozessor in der Mikrowelle läuft, selbst wenn ich beim letzten Mal was anderes gekocht habe als dieses Mal.

- Selbst wenn ich ein Mikrowellenentwickler wäre, und alle Dinge über die internen Abläufe kennen würde, ihre Software eingeschlossen, würde ich nicht darüber nachdenken, wenn ich die Mikrowelle benutze.

Das sind keine profunden Beobachtungen. Über so vieles können wir nicht gleichzeitig nachdenken. Um die Anzahl der Dinge zu reduzieren, mit denen wir uns beschäftigen müssen, arbeiten wir auf einem bestimmten Detailniveau.

Mit objektorientierten (OO) Begriffen ausgedrückt, wird der Level, auf dem wir arbeiten, als *Abstraktionslevel* bezeichnet. Wenn wir mit Nachos arbeiten, betrachte ich meine Mikrowelle als Box. So lange ich die Mikrowelle über ihr Interface verwende (die Knopfplatte), sollte nichts, was ich tue, die Mikrowelle dahin bringen, dass sie

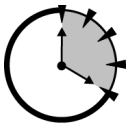
1. in einen inkonsistenten Zustand gerät und abstürzt
2. oder, was viel schlimmer wäre, mein Nacho in eine schwarze, brennende Masse verwandelt
3. oder, was am schlimmsten wäre, Feuer fängt und das Haus in Brand setzt.

17.1.1 Funktionale Nachos

Nehmen Sie an, ich sollte meinen Sohn bitten, einen Algorithmus zu schreiben, wie sein Vater Nachos macht. Nachdem er verstanden hätte, was ich von ihm will, würde er vermutlich etwas in der Art schreiben »öffne eine Dose Bohnen, reibe etwas Käse, schneide die Jalapenos« usw. Wenn es dann zu dem Teil mit der Mikrowelle käme, würde er etwas wie »fünf Minuten in der Mikrowelle kochen« schreiben.

Diese Beschreibung ist klar und vollständig. Aber es ist nicht die Art, in der ein funktionaler Programmierer ein Programm zur Herstellung von Nachos schreiben würde. Funktionale Programmierer leben in einer Welt, die frei ist von Objekten wie Mikrowellen und anderen Dingen. Sie kümmern sich um Flussdiagramme mit Myriaden von Pfaden. In einer funktionalen Lösung des Problems, Nachos herzustellen, würde die Kontrolle durch meinen Finger auf die Frontplatte fließen und dann in die internen Schaltungen der Mikrowelle. Dann würde die Kontrolle durch die komplexen logischen Pfade gehen, die dazu da sind, die Mikrowellenenergie anzuschalten und den Sound »fertig, hol mich raus« zu erzeugen.

In einer Welt wie dieser, ist es schwierig, von einem Abstraktionslevel zu sprechen. Es gibt keine Objekte und keine Abstraktion, hinter denen inhärente Komplexität versteckt werden könnte.



20 Min.

17.1.2 Objektorientierte Nachos

Bei objektorientierter Vorgehensweise, Nachos herzustellen, würden wir erst die Typen der Objekte in diesem Problem identifizieren: Chips, Bohnen, Käse und eine Mikrowelle. Dann würden wir mit der Aufgabe beginnen, diese Objekte in Software zu modellieren, ohne die Details zu berücksichtigen, wie sie im Programm verwendet werden.

Wenn wir Code auf Objektebene schreiben, arbeiten (und denken) wir auf einem Abstraktionsniveau der Basisobjekte. Wir müssen darüber nachdenken, eine nützliche Mikrowelle zu erstellen, wir müssen aber noch nicht über den logischen Prozess der Nacho-Zubereitung nachdenken. Schließlich haben die Designer der Mikrowelle auch nicht über das spezielle Problem nachgedacht, mir einen Snack zuzubereiten. Sie waren nur mit dem Problem befasst, eine nützliche Mikrowelle zu entwerfen und zu bauen.

Nachdem wir die Objekte erfolgreich codiert und getestet haben, können wir uns dem nächsten Abstraktionslevel zuwenden. Wir können auf dem Level der Nacho-Herstellung denken und nicht mehr auf Mikrowellenlevel. An dieser Stelle können wir die Anweisungen meines Sohnes leicht in C++-Code überführen.

Tatsächlich können wir die Leiter weiter hochsteigen. Der nächste Level nach oben könnte Dinge enthalten wie aufstehen, zur Arbeit gehen, nach Hause kommen, essen, ausruhen und schlafen, wobei der Verzehr von Nachos irgendwo in den Bereich von essen und ausruhen gehören würde.

17.2 Klassifizierung und Mikrowellen

Wesentlich im Konzept der Abstraktion ist die Klassifizierung. Wenn ich meinen Sohn fragen würde »Was ist eine Mikrowelle?« würde er wahrscheinlich sagen »Es ist Ofen, der ...« Wenn ich dann fragen würde »Was ist ein Ofen?« könnte er antworten »Es ist ein Gerät in der Küche, das ...« Ich könnte weitere Fragen stellen, bis wir schließlich bei der Antwort ankommen »Es ist ein Ding«, was eine andere Form von »Es ist ein Objekt« ist.

Mein Sohn versteht, dass unsere spezielle Mikrowelle ein Beispiel ist für den Typ der Dinge, die als Mikrowellen bezeichnet werden. Er versteht auch, dass die Mikrowelle ein spezieller Typ Ofen ist, und dieser wiederum ein spezielles Küchengerät ist.

Technisch gesagt, ist meine Mikrowelle eine *Instanz* der Klasse *Mikrowelle*. Die Klasse *Mikrowelle* ist eine *Unterklasse* der *Klasse Ofen*, und die Klasse Ofen ist eine *Superklasse* der Klasse *Mikrowelle*.

Menschen klassifizieren. Alles in unserer Welt ist in Klassen eingeteilt. Wir tun das, um die Anzahl der Dinge, die wir uns merken müssen, klein zu halten. Erinnern Sie sich z.B. daran, wann Sie zum ersten Mal den Ford-basierten Jaguar (oder den neuen Neon) gesehen haben. Die Werbung nannte den Jaguar »revolutionär, eine neue Art Auto«. Aber Sie und ich wissen, dass das nicht stimmt. Ich mag das Aussehen des Jaguars – ich mag es sogar sehr – aber es ist nur ein Auto. Als solches teilt es alle (oder zumindest die meisten) Eigenschaften mit anderen Autos. Er hat ein Lenkrad, Sitze, einen Motor, Bremsen, usw. Ich wette, ich könnte sogar einen Jaguar ohne Hilfe fahren.

Ich will den wenigen Platz, den ich in diesem Buch zur Verfügung habe, nicht mit all den Dingen verschwenden, die ein Jaguar mit anderen Autos gleich hat. Ich muss nur an den Satz »ein Jaguar ist ein Auto, das ...« denken und die wenigen Dinge, die einzigartig für Jaguar sind. Autos sind eine Unterklasse von bereiften Fahrzeugen, von denen es auch andere gibt, wie z.B. LKW und Pickups. Vielleicht sind bereifte Fahrzeuge eine Unterklasse von Fahrzeuges, die auch Boote und Flugzeuge einschließt. Das lässt sich beliebig fortsetzen.

17.2.1 Warum solche Objekte bilden?

Es scheint leichter zu sein, eine Mikrowelle zu entwerfen und zu bauen, speziell für unser eines Problem als ein separates, allgemeineres Ofenobjekt. Stellen Sie sich z.B. vor, ich wollte eine Mikrowelle bauen, um Nachos, und nur Nachos, darin zuzubereiten. Diese Mikrowelle bräuchte dann keine Frontplatte, sondern nur einen Startknopf. Sie kochen Nachos immer gleich lang. Wir könnten uns

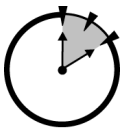
den ganzen Unsinn z.B. zum Auftauen, sparen. Die Mikrowelle könnte winzig sein. Sie müsste nur einen kleinen Teller aufnehmen können. Nur sehr wenig Platz würde so für Nachos verschwendet.

In diesem Sinne, lassen Sie uns das Konzept »Mikrowellenofen« vergessen, Alles, was wir wirklich brauchen, sind die Interna eines Ofens. In einem Rezept fassen wir dann alle Instruktionen zusammen, um ihn zum Laufen zu bringen: »Legen Sie die Nachos in die Box. Verbinden Sie das rote und das schwarze Kabel. Sie hören ein leichtes Summen«. So etwas in der Art.

Wie dem auch sei, der funktionale Ansatz hat einige Probleme:

- **Zu komplex.** Wir wollen die Details des Ofenbauens nicht vermischen mit den Details der Nachozubereitung. Wenn wir die Details nicht finden und aus der Fülle der Details herausziehen können, um sie separat zu bearbeiten, müssen wir immer mit der Gesamtkomplexität des Problems arbeiten.
- **Nicht flexibel.** Wenn wir den Mikrowellenofen durch einen anderen Typ Ofen ersetzen müssen, könnten wir das tun, solange der neue Ofen das gleiche Interface wie der alte Ofen hat. Ohne ein klar definiertes Interface wird es unmöglich, einen Objekttyp sauber zu entfernen und durch einen anderen zu ersetzen.
- **Nicht wiederverwendbar.** Öfen werden verwendet, um verschiedene Gerichte zuzubereiten. Wir müssen nicht jedes Mal einen neuen Ofen kreieren, wenn wir ein neues Rezept haben. Wenn ein Problem gelöst ist, wäre es schön, die Lösung auch in zukünftigen Programmen wiederverwenden zu können.

Es kostet mehr, ein generisches Objekt zu schreiben. Es wäre billiger, eine Mikrowelle nur für Nachos zu bauen. Wir könnten auf die teure Zeituhr und Knöpfe verzichten, die für Nachos nicht benötigt werden. Nachdem wir ein generisches Objekt in mehr als einer Anwendung verwendet haben, kommt der Vorteil einer etwas teureren generischen Klasse gegenüber der wiederholten Entwicklung billigerer Klassen für jede neue Applikation zum Tragen.



10 Min.

17.2.2 Selbstenthaltende Klassen

Lassen Sie uns reflektieren, was wir gelernt haben. Im objektorientierten Zugang der Programmierung

- identifiziert der Programmierer die Klassen, die benötigt werden, um ein Problem zu lösen. Ich wusste gleich, dass ich einen Ofen brauche, um Nachos zuzubereiten.
- Der Programmierer erzeugt selbstenthaltende Klassen, die den Anforderungen des Problems entsprechen, und er kümmert sich nicht um die Details der Gesamtanwendung.
- Der Programmierer schreibt die Anwendung unter Verwendung der gerade erzeugten Klassen, ohne darüber nachzudenken, wie sie intern funktionieren.

Ein integraler Bestandteil dieses Programmiermodells ist, dass eine Klasse für sich selber verantwortlich ist. Eine Klasse sollte jederzeit in einem definierten Zustand sein. Es sollte nicht möglich sein, das Programm zum Absturz zu bringen, indem man einer Klasse ungültige Daten oder eine ungültige Folge gültiger Daten übergibt.

Paradigma ist ein anderes Wort für Programmiermodell.

Viele der Features von C++, die in den folgende Kapiteln beschrieben werden, behandeln die Fähigkeit von Klassen, sich selber gegen fehlerhafte Programme zu schützen, die nur auf einen Absturz warten.

Zusammenfassung

In dieser Sitzung haben Sie die fundamentalen Konzepte der objektorientierten Programmierung gesehen.

0 Min.

- Objektorientierte Programme bestehen aus locker verbundenen Objekten.
- Jede Klasse repräsentiert ein Konzept der realen Welt.
- Objektorientierte Klassen werden so geschrieben, dass sie in gewisser Hinsicht unabhängig sind von dem Programm, das sie benutzt.
- Objektorientierte Klassen sind verantwortlich für ihr eigenes Wohlbefinden.

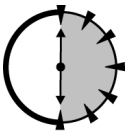
Selbsttest

1. Was bedeutet der Begriff `Abstraktionslevel`? (Siehe »Abstraktion und Mikrowellen«)
2. Was ist mit `Klassifizierung` gemeint? (Siehe »Klassifizierung und Mikrowellen«)
3. Was sind die drei fundamentalen Probleme des funktionalen Programmiermodells, die vom objektorientierten Programmiermodell zu lösen versucht werden? (Siehe »Warum solche Objekte bilden?«)
4. Wie stellt man Nachos her? (Siehe »Siehe »Abstraktion und Mikrowellen«)

Aktive Klassen

Checkliste

- ☑ Klassen durch Elementfunktionen in aktive Agenten verwandeln
- ☑ Elementfunktionen mit Namen versehen
- ☑ Elementfunktionen innerhalb und außerhalb der Klasse definieren
- ☑ Elementfunktionen aufrufen
- ☑ Klassendefinitionen in `#include`-Dateien sammeln
- ☑ Auf `this` zugreifen



30 Min.

Objekte in der realen Welt sind unabhängige Agenten (abgesehen von ihrer Abhängigkeit von Strom, Luft usw.). Eine Klasse sollte so unabhängig wie möglich sein. Es ist für eine struct-Struktur unmöglich, von ihrer Umgebung unabhängig zu sein. Die Funktionen, die ihre Daten manipulieren, müssen außerhalb der Struktur definiert sein. Aktive Klassen haben die Fähigkeit, diese Manipulationsfunktionen in sich selbst zu bündeln.

18.1 Klassenrückblick

Ein Klasse ermöglicht die Gruppierung von verwandten Datenelement in einer Einheit. Z.B. könnten wir eine Klasse `Student` wie folgt definieren:

```
class Student
{
  public:
    int   nSemesterHours;   // Semesterstunden bis
                           // zur Graduierung
    float dAverage;         // mittlerer Grad
};
```

Jede Instanz von `Student` enthält ihre eigenen zwei Datenelemente:

```
void fn(void)
{
    Student s1;
    Student s2;
    s1.nSemesterHours = 1;  // ist nicht gleich ...
    s2.nSemesterHours = 2;  // ... mit diesem
}
```

Die beiden `nSemesterHours` sind verschieden, weil sie zu verschiedenen Studenten gehören (s1 und s2).

Eine Klasse mit nichts als Datenelementen wird auch als *Struktur* bezeichnet und wird über das Schlüsselwort `struct` definiert, das von der Programmiersprache C herkommt. Eine `struct` ist identisch mit einer Klasse, außer dass das Schlüsselwort `public` nicht benötigt wird.

Es ist möglich, Zeiger auf Klassenobjekte zu definieren, und diese Objekte vom Heap zu allozieren, wie das folgende Beispiel zeigt:

```
void fn()
{
    Student* pS1 = new Student;
    pS1->nSemesterHours = 1;
}
```

Es ist möglich, Klassenobjekte an Funktionen zu übergeben:

```
void studentFunc(Student s);
void studentFunc(Student* pS);

void fn()
{
    Student s1 = {12, 4.0};
    Student* pS = new Student;

    studentFunc (s1); // ruft studentFunc(Student)
    studentFunc (ps); // ruft studentFunc(Student*)
}
```


18.2 Grenzen von struct

Klassen, die nur Datenelemente enthalten, haben entscheidende Grenzen.

Programme existieren in der realen Welt. D.h., jedes nicht triviale Programm ist dafür entworfen, eine Funktion der realen Welt bereitzustellen. Meistens, aber nicht immer, haben diese Funktionen eine Analogie, die manuell ausgeführt werden könnte. Z.B. könnte ein Programm den mittleren Grad eines Studenten ausrechnen. Das könnte per Hand mit Papier und Bleistift durchgeführt werden, es ist aber elektronisch viel einfacher und schneller.

Je genauer ein Programm das Leben widerspiegelt, um so einfacher ist das Programm zu verstehen. Wenn es einen Typ Ding gibt, das »Student« genannt wird, dann wäre es schön, eine Klasse `Student` zu haben, die alle wesentlichen Eigenschaften von Studenten besitzt. Eine Instanz der Klasse `Student` würde dann einem Studenten entsprechen.

Eine kleine Klasse `Student`, die alle Informationen beschreibt, die zur Berechnung des mittleren Grades eines Studenten benötigt werden, finden Sie am Anfang dieser Sitzung. Das Problem mit

dieser Klasse ist, dass sie nur passive Eigenschaften von Studenten enthält. D.h., ein Student hat eine Eigenschaft für die Anzahl der Semester und seinen mittleren Grad. (Ein Student hat auch einen Namen, eine Sozialversicherungsnummer usw., aber es ist in Ordnung, diese Eigenschaften wegzulassen, wenn sie nicht zu dem Problem gehören, das wir lösen möchten.) Studenten beginnen Vorlesungen, brechen Vorlesungen ab und beenden Vorlesungen erfolgreich. Das sind aktive Eigenschaften der Klasse.

18.2.1 Eine funktionale Lösung

Sicher, es ist möglich, einer Klasse aktive Eigenschaften zuzufügen, indem man eine Menge von Funktionen für die Klasse schreibt:

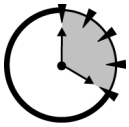
```
// Definiere eine Klasse, um die passiven
// Eigenschaften eines Studenten zu speichern
class Student
{
  public:
    int   nSemesterHours;   // Semesterstunden bis
                           // zur Graduierung
    float dAverage;         // mittlerer Grad
};

// eine Kursklasse
class Course
{
  public:
    char* pszName;
    int   nCourseNumber;
    int   nNumHours;
};

// definiere eine Menge von Funktionen, um die
// aktiven Eigenschaften eines Studenten zu
// beschreiben
void startCourse(Student* pS, Course* pC);
void dropCourse(Student* pS, int nCourseNumber);
void completeCourse(Student* pS, int nCourseNumber);
```

Diese Lösung funktioniert – in der Tat ist das die Lösung in nicht objektorientierten Programmiersprachen wie C. Es gibt jedoch ein Problem mit dieser Lösung.

In der Art und Weise, in der dieser Auszug geschrieben ist, hat Student nur passive Eigenschaften. Es gibt da ein nebulöses »Ding«, das aktive Agenten hat, wie `startCourse( )`, das auf Objekten aus der Klasse `Student` arbeitet (und Objekten aus der Klasse `Course`). Dieses nebulöse Ding hat keine eigenen Dateneigenschaften. Obwohl sie funktioniert, spiegelt diese Beschreibung nicht die Realität wider.

Wir würden gerne die aktiven Eigenschaften aus diesem undefinierten Ding herausnehmen und sie der Klasse `Student` selber hinzufügen, so dass jede Instanz der Klasse `Student` mit einem kompletten Satz von Eigenschaften ausgerüstet ist.

Das Hinzufügen aktiver Eigenschaften zu einer Klasse, statt diese aus der Klasse herauszulassen, scheint Ihnen vielleicht nebensächlich zu sein, es wird aber im weiteren Verlauf dieses Buches immer wichtiger.



18.3 Definition einer aktiven Klasse

20 Min.

Die aktiven Eigenschaften eines Studenten könnten wie folgt der Klasse `Student` zugefügt werden:

```
class Student
{
  public:
    // die passiven Eigenschaften der Klasse
    int   nSemesterHours;   // Semesterstunden bis
                           // zur Graduierung
    float dAverage;         // mittlerer Grad

    // die aktiven Eigenschaften der Klasse
    float startCourse(Course*);
    void  dropCourse(int nCourseNumber);
    void  completeCourse(int nCourseNumber);
};
```

Die Funktion `startCourse(Course*)` ist eine Eigenschaft der Klasse, wie `nSemesterHours` und dAverage.

Eine Funktion, die Element einer Klasse ist, wird Elementfunktion genannt. Aus historischen Gründen, die nur sehr wenig mit C++ zu tun haben, werden Elementfunktionen auch Methoden genannt. Wahrscheinlich weil der Begriff »Methode« der verwirrendere der beiden Begriffe ist, wird er am häufigsten verwendet.

Es gibt keinen Namen für Funktionen oder Daten, die nicht Element einer Klasse sind. Ich bezeichne sie als Nichtelemente. Alle Funktionen, die sie bisher gesehen haben, sind Nichtelemente gewesen, weil sie nicht zu einer Klasse gehören.

C++ kümmert sich nicht um die Ordnung der Elementfunktionen innerhalb einer Klasse. Die Datenelemente können vor oder nach den Elementfunktionen stehen, sie können auch miteinander vermischt werden. Ich selber bevorzuge es, die Datenelemente vor die Elementfunktionen zu platzieren.

18.3.1 Namengebung für Elementfunktionen

Der vollständige Name der Funktion `startCourse(Course*)` ist Student::`startCourse(Course*)`. Der Klassenname am Anfang zeigt an, dass die Funktion ein Element der Klasse Student ist. (Der Klassenname wird dem erweiterten Namen der Funktion zugefügt, wie die Argumente zum Namen einer überladenen Funktion hinzugefügt wurden.) Wir könnten weitere Funktionen mit Namen `startCourse( )` haben, die Elemente in anderen Klassen wären, wie z.B. Teacher::`startCourse( )`. Eine Funktion `startCourse( )` ohne einen Klassennamen am Anfang, ist eine herkömmliche Nichtelementfunktion.

Tatsächlich ist der vollständige Name der Nichtelementfunktion `addCourse( )` ***gleich*** `::addCourse( )`***. Die beiden Doppelpunkte :: ohne Klassennamen auf ihrer linken Seite weisen ausdrücklich darauf hin, dass es sich um eine Nichtelementfunktion handelt.***

Datenelemente verhalten sich nicht anders als Elementfunktionen in Bezug auf erweiterte Namen. Außerhalb einer Struktur ist es nicht ausreichend, nur `nSemesterHours` zu verwenden. Das Datenelement `nSemesterHours` macht nur im Kontext der Klasse `Student` Sinn. Der erweiterte Name von nSemesterHours ist Student::`nSemesterHours`.

Der Operator :: wird auch *Bereichsauflösungsoperator* genannt, weil er die Klasse identifiziert, zu der ein Element gehört. Der Operator :: kann mit einer Nichtelementfunktion und mit einem leeren Strukturnamen aufgerufen werden. Die Nichtelementfunktion `startCourse( )` sollte als `::startCourse( )` angesprochen werden.

Der Operator ist optional, außer wenn es zwei Funktionen mit gleichem Namen gibt. Hier ein Beispiel dazu:

```
float startCourse(Course*);

class Student
{
  public:
    int   nSemesterHours;  // Semesterstunden bis zur
                           // Graduierung
    float dAverage;        // mittlerer Grad

    // beginne neuen Kurs
    float startCourse(Course* pCourse)
    {
      // ... was auch immer ...

      startCourse(pCourse); // globale Funktion(?)

      // ... weitere Anweisungen ...
   }
};
```

Wir wollen, dass die Elementfunktion `Student::startCourse( )` die Nichtelementfunktion `::startCourse( )` aufruft. Ohne den Operator :: bezieht sich ein Aufruf von `startCourse( )` jedoch auf Student::`startCourse( )`. Das führt dazu, dass sich die Funktion selber aufruft. Das

Hinzufügen des Operators :: an den Anfang des Aufrufs leitet den Aufruf wie gewünscht an die globale Funktion weiter:

```
class Student
{
  public:
    int   nSemesterHours;  // Semesterstunden bis zur
                           // Graduierung
    float dAverage;        // mittlerer Grad

   // starte neuen Kurs
   float startCourse(Course* pCourse)
   {
       ::startCourse(pCourse); // globale Funktion
   }
};
```

Der vollständig erweiterte Name einer Nichtelementfunktion enthält also nicht nur die Argumente, wie wir in Sitzung 9 gesehen haben, sondern auch den Namen der Klasse, zu der die Funktion gehört – der ist leer für Nichtelementfunktionen.

18.4 Definition einer Elementfunktion einer Klasse

Eine Elementfunktion kann entweder in der Klasse oder separat definiert werden. Betrachten Sie die folgende Definition der Methode `addCourse(int, float)` innerhalb der Klasse:

```
class Student
{
  public:
    int   nSemesterHours;  // Semesterstunden bis zur
                           // Graduierung
    float dAverage;        // mittlerer Grad

    // füge einen absolvierten Kurs ein
    float addCourse(int hours, float grade)
    {
        float weighted;
        weighted = nSemesterHours * dAverage;

        // füge den neuen Kurs ein
        nSemesterHours += hours;
        weighted += grade * hours;
        dAverage = weighted / nSemesterHours;
        return dAverage;
    }
};
```

Der Code von `addCourse(int, float)` sieht nicht anders aus als der jeder anderen Funktion, außer dass er eingebettet ist in die Klasse.

Elementfunktionen, die innerhalb der Klasse definiert sind, werden standardmäßig als Inline-Funktionen behandelt (s.u.). Hauptsächlich ist dies der Fall, weil Elementfunktionen, die in der Klasse definiert sind, sehr kurz sind, und kurze Funktionen die primären Kandidaten für eine Behandlung als Inline-Funktion sind.

Inline-Funktionen

Normalerweise veranlasst eine Funktionsdefinition den C++-Compiler, Maschinencode an einer bestimmten Stelle des ausführbaren Programms zu platzieren. Jedes Mal, wenn die Funktion aufgerufen wird, fügt C++ eine Art Sprung an die Stelle ein, an der die Funktion gespeichert ist. Wenn die Funktion durchlaufen wurde, geht die Kontrolle zurück zu dem Punk, von wo aus die Funktion aufgerufen wurde.
C+ definiert einen speziellen Typ Funktion, der als *Inline*-Funktion bezeichnet wird. Wenn eine Inline-Funktion aufgerufen wird, generiert der C++-Compiler den Code direkt an dieser Stelle. Jeder Aufruf der Inline-Funktion erhält seine eigene Kopie des Maschinencodes.
Inline-Funktionen werden schneller ausgeführt, weil der Computer nicht an eine andere Stelle springen und Initialisierungen ausführen muss, um die Verarbeitung fortzufahren. Denken Sie jedoch daran, dass Inline-Funktionen mehr Speicher benötige. Wenn eine Inline-Funktion zehnmal aufgerufen wird, befinden sich 10 Kopien des Maschinencodes im Speicher.
Weil der Unterschied zwischen Inline-Funktionen und konventionellen Funktionen, die manchmal auch als Outline-Funktionen bezeichnet werden, in der Ausführungsgeschwindigkeit klein ist, sind nur kleine Funktionen Kandidaten für eine Inline-Behandlung. Zusätzlich zwingen bestimmte Konstruktionen eine Inline-Funktion zu einer Behandlung als Outline-Funktion.

18.5 Schreiben von Elementfunktionen außerhalb einer Klasse

Für längere Funktionen führt das direkte Einfügen des Funktionscodes zu sehr langen und schwerfälligen Klassendefinitionen. Um dies zu verhindern, erlaubt es uns C++, Elementfunktionen außerhalb der Klasse zu definieren.

Unsere Methode `addCourse( )`, die außerhalb der Klasse definiert ist, sieht dann so aus:

```
class Student
{
  public:
    int   nSemesterHours;  // Semesterstunden bis zur
                           // Graduierung
    float dAverage;        // mittlerer Grad

    // füge einen absolvierten Kurs ein
    float addCourse(int hours, float grade);
};
float Student::addCourse(int hours, float grade)
{
    float weighted;
    weighted = nSemesterHours * dAverage;

    // füge den neuen Kurs ein
    nSemesterHours += hours;
    weighted += grade * hours;
    dAverage = weighted / nSemesterHours;
    return dAverage;
}
```

Hier sehen wir, dass die Klassendefinition nichts weiter als eine *Prototypdeklaration* der Funktion `addCourse( )` enthält. Die tatsächliche *Funktionsdefinition* steht separat.

Eine Deklaration definiert den Typ einer Sache. Eine Definition definiert den Inhalt einer Sache.

Die Analogie mit einer Prototypdeklaration ist exakt. Die Deklaration in der Struktur ist eine Prototypdeklaration und ist, wie alle Prototypdeklarationen, notwendig.

Als die Funktion innerhalb der Klasse `Student` definiert wurde, war es nicht nötig, dass der Klassenname im Namen der Funktion enthalten ist; es wurde der Name der enthaltenden Klasse angenommen. Wenn die Funktion alleine steht, wird der Klassenname benötigt. Es ist wie bei mir zu Hause. Meine Frau ruft mich nur bei meinem Vornamen (vorausgesetzt ich bin nicht in der Hundehütte). Innerhalb der Familie wird der Nachname angenommen. Außerhalb der Familie (und meinem Bekanntenkreis) rufen mich andere mit meinem vollen Namen.

18.5.1 Include-Dateien

Es ist üblich, Klassendefinitionen und Funktionsprototypen in eine Datei zu schreiben, die die Endung .h trägt, getrennt von der .cpp-Datei, die die tatsächlichen Funktionsdefinitionen enthält. Die .h-Datei wird dann in der .cpp-Datei »included« (=eingebunden), wie im Folgenden zu sehen ist.

Die Datei student.h wird am besten so definiert:

```
class Student
{
  public:
    int   nSemesterHours;  // Semesterstunden bis zur
                           // Graduierung
    float dAverage;        // mittlerer Grad

    // füge einen absolvierten Kurs ein
    float addCourse(int hours, float grade);
};
```

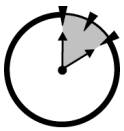
Die Datei student.cpp sieht wie folgt aus:

```
#include »student.h«;
float Student::addCourse(int hours, float grade)
{
    float weighted;
    weighted = nSemesterHours * dAverage;

    // füge den neuen Kurs ein
    nSemesterHours += hours;
    weighted += grade * hours;
    dAverage = weighted / nSemesterHours;
    return dAverage;
}
```

Die Direktive `#include` besagt »ersetze diese Direktive durch den Inhalt der Datei student.h«.

Die Direktive `#include` hat nicht das Format einer C++-Anweisung, weil es von einem separaten Interpreter verarbeitet wird, der vor dem C++-Compiler ausgeführt wird.

Klassendefinitionen und Funktionsprototypen in Include-Dateien zu packen, ermöglichen es mehreren C++-Modulen, die gleichen Informationen einzubinden, ohne sie wiederholen zu müssen. Das reduziert den Aufwand und, was noch wichtiger ist, es reduziert die Wahrscheinlichkeit, dass mehrere Quelldateien nicht mehr synchron sind.



18.6 Aufruf einer Elementfunktion

Bevor wir uns ansehen, wie Elementfunktionen aufgerufen werden, lassen Sie uns daran erinnern, wie ein Datenelement referenziert wird:

10 Min.

```
#include »student.h«
Student s;
void fn(void)
{
    // Zugriff auf Datenelemente von s
    s.nSemesterHours = 10;
    s.dAverage       = 3.0;
}
```

Wir müssen ein Objekt zusammen mit dem Elementnamen spezifizieren, wenn wir ein Objektelement ansprechen wollen. In anderen Worten, das Folgende macht keinen Sinn:

```
#include »student.h«
void fn(void)
{
    Student s;

    // Zugriff auf die Datenelemente von s
    // keiner der Zugriffe ist zulässig
    nSemesterHours = 10;  // Element welches Objekt
                          // aus welcher Klasse?
    Student::nSemesterHours = 10;
                          // ok, ich weiß die Klasse,
                          // aber ich weiß nicht
                          // welches Objekt
     s.nSemesterHours = 10; // das ist in Ordnung
}
```

Elementfunktionen werden mit einem Objekt aufgerufen, wie Datenelemente angesprochen werden:

```
void fn()
{
    Student s;

    // referenziere die Datenelemente der Klasse
    s.nSemesterHours = 10;
    s.dAverage       = 3.0;

    // Zugriff auf die Elementfunktion
    s.addCourse(3, 4.0);
}
```

Eine Elementfunktion ohne ein Objekt aufzurufen, macht nicht mehr Sinn, als ein Datenelement ohne ein Objekt anzusprechen. Die Syntax für den Aufruf einer Elementfunktion sieht aus wie eine Kreuzung der Syntax für Zugriffe auf Datenelemente und dem Aufruf konventioneller Funktionen.

18.6.1 Aufruf einer Elementfunktion über einen Zeiger

Die gleiche Parallele wie für die Objekte selber kann auch für Zeiger auf Objekte gezogen werden. Das Folgende referenziert ein Datenelement eines Objektes über einen Zeiger:

```
#include »student.h«

void someFn(Student *pS)
{
    // Zugriff auf die Datenelemente der Klasse
    pS->nSemesterHours = 10;
    pS->dAverage       = 3.0;

    // Zugriff auf die Elementfunktion
    // (d.h. Aufruf dieser Funktion)
    pS->addCourse(3, 4.0);
}

int main()
{
    Student s;

    someFn(&s);
    return 0;
}
```

Eine Elementfunktion mit einer Referenz auf ein Objekt aufzurufen, erscheint identisch mit der Benutzung des Objektes selber. Erinnern Sie sich daran, dass bei der Übergabe oder Rückgabe einer Referenz als Argument einer Funktion C++ nur die Adresse des Objektes übergibt. Bei Verwendung einer Referenz dereferenziert C++ die Adresse automatisch, wie das folgende Beispiel zeigt:

```
#include »student.h«

// das Gleiche wie eben, nur dieses
// Mal mit Referenzen
void someFn(Student &refS)
{
    refS.nSemesterHours = 10;
    refS.dAverage       = 3.0;
    refS.addCourse(3, 4.0); // Aufruf Elementfunktion
}

Student s;
int main()
{
    someFn(s);
    return 0;
}
```

18.6.2 Zugriff auf andere Elemente von einer Elementfunktion aus

Es ist klar, warum Sie keine Elemente einer Klasse ohne ein Objekt ansprechen können. Sie müssen z.B. wissen, welches `dAverage` von welchem `Student`-Objekt? Schauen Sie sich nochmals die Definition der Elementfunktion `Student::addCourse( )` an. Diese Funktion greift auf Klassenelemente zu, ohne explizit ein Objekt zu referenzieren, was in direktem Widerspruch dazu steht, was ich gerade gesagt habe.

Sie können ein Element einer Klasse nicht ohne ein Objekt ansprechen; innerhalb einer Elementfunktion jedoch wird das Objekt als das referenzierte Objekt angenommen. Es ist leichter, das an einem Beispiel zu sehen:

```
#include »student.h«
float Student::addCourse(int hours, float grade)
{
    float weighted;
    weighted = nSemesterHours * dAverage;

    // füge den neuen Kurs ein
    nSemesterHours += hours;
    weighted += hours * grade;
    dAverage = weighted / nSemesterHours;
    return dAverage;
}

int main(int nArgs, char* pArgs[])
{
    Student s;
    Student t;

    s.addCourse(3, 4.0);  // Note 4
    t.addCourse(3, 2.5);  // Note 2-
    return 0;
}
```

Wenn `addCourse( )` mit dem Objekt s aufgerufen wird, beziehen sich alle anderen unqualifizierten Elemente innerhalb von `addCourse( )` auf s. Somit wird `nSemesterHours` zu `s.nSemesterHours`, `dAverage` wird zu `s.dAverage`. Im Aufruf `t.addCourse( )` in der nächsten Zeile beziehen sich die gleichen Referenzen auf `t.nSemesterHours` und `t.dAverage`.

Das Objekt, mit dem die Elementfunktion aufgerufen wird, ist das »aktuelle« Objekt, und alle unqualifizierten Referenzen auf Klassenelemente beziehen sich auf dieses Objekt. Oder anders gesagt, beziehen sich unqualifizierte Referenzen auf Klassenelemente auf das aktuelle Objekt.

Wie wissen Elementfunktionen, was das aktuelle Objekt ist? Es ist keine Magie – die Adresse des Objektes wird an die Elementfunktion übergeben als implizites und nicht sichtbares erstes Argument. In anderen Worten findet die folgende Konvertierung statt:

```
s.addCourse(3, 2.5)
```

wird zu

```
Student::addCourse(&s, 3, 2.5);
```

(Sie können diese interpretative Syntax so nicht verwenden; sie ist nur ein Weg, um zu verstehen, was C++ tut.)

Innerhalb der Funktion hat dieser implizite Zeiger auf das aktuelle Objekt einen Namen, für den Fall, dass Sie darauf zugreifen müssen. Dieser versteckte Objektzeiger heißt `this` (=dieses). Der Typ von `this` ist immer ein Zeiger auf ein Objekt der entsprechenden Klasse. Innerhalb der Klasse `Student` ist `this` vom Typ `Student*`.

Jedes Mal, wenn eine Elementfunktion auf ein anderes Element der gleichen Klasse verweist, ohne explizit ein Objekt zu benennen, nimmt C++ an, dass `this` gemeint ist. Sie können this auch explizit verwenden. Wir hätten `Student::addCourse( )` auch so schreiben können:

```
#include »student.h«
float Student::addCourse(int hours, float grade)
{
    float weighted;

   // referenziere 'this' explicit
   weighted = this->nSemesterHours * this->dAverage;

   // gleiche Rechnung mit 'this' implizit
   weighted = nSemesterHours * dAverage;

   // füge den neuen Kurs ein
   this->nSemesterHours += hours;
   weighted += hours * grade;
   this->dAverage = weighted / this->nSemesterHours;
   return this->dAverage;
}
```

Ob wir this explizit hinschreiben oder es implizit lassen, wie wir es bereits getan haben, hat den gleichen Effekt.

18.7 Überladen von Elementfunktionen

Elementfunktionen können überladen werden in der gleichen Weise wie konventionelle Funktionen. Erinnern Sie sich jedoch daran, dass der Klassenname Teil des erweiterten Namens ist. Somit sind die folgenden Funktionen legal:

```
class Student
{
  public:
    // grade - aktueller mittlerer Grad
    float grade();

    // grade - setze Grad und gib vorigen Wert zurück
    float grade(float newGrade);

    // ... Datenelemente und alles weitere ...
};

class Slope
{
  public:
    // grade - Grad des Gefälles
    float grade();

    // ... hier steht der Rest ...
};

// grade - gibt Zeichenäquivalent eines Wertes zurück
char grade(float value);

int main(int nArgs, char* pArgs[])
{
    Student s;
    Slope o;

    // rufe verschiedene Varianten von grade()auf
    s.grade(3.5);        // Student::grade(float)
    float v = s.grade(); // Student::grade()
    char c = grade(v);   // ::grade(float)
    float m = o.grade(); // Slope::grade()
    return 0;
}
```

Jeder Aufruf von `main( )` aus ist im Kommentar mit dem erweiterten Namen der aufgerufenen Funktion versehen.

Wenn überladene Funktionen aufgerufen werden, werden sowohl die Argumente der Funktion, und der Typ des Objektes (falls vorhanden), mit dem die Funktion aufgerufen wird, verwendet, um den Aufruf unzweideutig zu machen.

Der Begriff `unzweideutig` ist ein objektorientierter Begriff, der sagen soll »entscheide zur Compilezeit, welche überladene Funktion aufgerufen werden soll«. Wir können auch sagen, dass die Aufrufe `aufgelöst` werden.

0 Min.

Im Beispielcode unterscheiden sich die beiden ersten Aufrufe der Elementfunktionen `Student:: grade(float)` und `Student::grade( )` durch ihre Argumentlisten. Der dritte Aufruf hat kein Objekt, somit bezeichnet es unzweideutig die Nichtelementfunktion `grade(float)`. Weil der letzte Aufruf mit einem Objekt vom Typ `Slope` durchgeführt wird, muss er sich auf die Elementfunktion `Slope::grade( )` beziehen.

Zusammenfassung

Je näher Sie das Problem, das Sie mit C++ lösen wollen, modellieren können, desto leichter kann das Problem gelöst werden. Diejenigen Klassen, die nur Datenelemente enthalten, können nur passive Eigenschaften von Objekten modellieren. Das Hinzufügen von Elementfunktionen macht die Klasse mehr zu einem Objekt wie in der realen Welt, in dem Sinne, dass es nun auf die Welt »außen«, d.h. den Rest des Programms, reagieren kann. Außerdem kann die Klasse für ihr eigenes Wohlergehen verantwortlich gemacht werden, in der gleichen Weise, wie Objekte in der realen Welt sich selbst schützen.

- Elemente einer Klasse können Funktionen oder Daten sein. Solche Elementfunktionen machen eine Klasse aktiv. Der vollständige Name einer Elementfunktion schließt den Namen der Klasse ein.
- Elementfunktionen können entweder innerhalb oder außerhalb der Klasse definiert werden. Elementfunktionen, die außerhalb einer Klasse definiert sind, sind der Klasse schwerer zuzuordnen, tragen aber zu einer besseren Lesbarkeit der Klasse bei.
- Innerhalb einer Elementfunktion kann das aktuelle Objekt über das Schlüsselwort `this` referenziert werden.

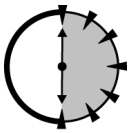
Selbsttest

1. Was ist falsch daran, Funktionen außerhalb einer Klasse zu deklarieren, die Datenelemente der Klasse direkt manipulieren? (Siehe »Eine funktionale Lösung«)
2. Eine Funktion, die ein Element einer Klasse ist, wird wie bezeichnet? Es gibt zwei Antworten auf diese Frage. (Siehe »Definition einer aktiven Klasse«)
3. Beschreiben Sie die Bedeutung der Reihenfolge von Funktionen innerhalb einer Klasse. (Siehe »Definition einer aktiven Klasse«)
4. Wenn eine Klasse `X` ein Element `Y(int)` besitzt, was ist der vollständige Name der Funktion? (Siehe »Schreiben von Elementfunktionen außerhalb einer Klasse«)
5. Warum wird eine Include-Datei so genannt? (Siehe »Include-Dateien«)

Erhalten der Klassenintegrität

Checkliste

- ☑ Einen Konstruktor schreiben und benutzen
- ☑ Datenelemente konstruieren
- ☑ Einen Destruktor schreiben und benutzen
- ☑ Zugriff auf Datenelemente kontrollieren



30 Min.

Ein Objekt kann nicht für sein Wohlergehen verantwortlich gemacht werden, wenn es keine Kontrolle darüber hat, wie es erzeugt und verwendet wird. In dieser Sitzung untersuchen wir die Möglichkeiten in C++, die Integrität von Objekten zu erhalten.

19.1 Erzeugen und Vernichten von Objekten

C++ kann ein Objekt als Teil einer Deklaration initialisieren, z.B.:

```
class Student
{
  public:
    int   nSemesterHours;
    float dAverage;
};

void fn()
{
    Student s = {0, 0};
    //... Fortsetzung der Funktion ...
}
```

Hierbei hat `fn( )` totale Kontrolle über das `Student`-Objekt.

Wir könnten die Klasse mit einer Initialisierungsfunktion ausstatten, die von der Anwendung aufgerufen wird, sobald ein Objekt erzeugt wird. Das gibt der Klasse die Kontrolle darüber, wie ihre Datenelemente initialisiert werden. Die Lösung sieht dann so aus:

```
class Student
{
  public:
    // data members
    int   nSemesterHours;
    float dAverage;

    // Elementfunktionen
    // init - initialisiere ein Objekt mit einem
    //        gültigen Zustand
    void init()
    {
        semesterHours = 0;
        dAverage = 0.0;
    }
};

void fn()
{
    // erzeuge gültiges Student-Objekt
    Student s;   // erzeuge das Objekt...
    s.init();    // ... und initialisiere es

    //... Fortsetzung der Funktion ...
}
```

Das Problem mit der »init«-Lösung ist, dass sich die Klasse auf die Anwendung verlassen muss, dass die Funktion `init( )` aufgerufen wird. Das ist nicht die angestrebte Lösung. Was wir eigentlich wollen, ist ein Mechanismus, der ein Objekt automatisch initialisiert, wenn es erzeugt wird.

19.1.1 Der Konstruktor

C++ ermöglicht es einer Klasse, ihre Objekte zu initialisieren, indem eine spezielle Funktion bereitgestellt wird. Diese wird als Konstruktor bezeichnet.

Ein Konstruktor ist eine Elementfunktion, die automatisch aufgerufen wird, wenn ein Objekt erzeugt wird. In gleicher Weise wird ein Destruktor aufgerufen, wenn ein Objekt vernichtet wird.

C++ führt einen Aufruf eines Konstruktors immer aus, wenn ein Objekt erzeugt wird. Der Konstruktor trägt den gleichen Namen wie die Klasse. Auf diese Weise weiß der Compiler, welche Elementfunktion ein Konstruktor ist.

Die Entwickler von C++ hätten auch eine andere Regel aufstellen können, z.B. »Der Konstruktor muss `init( )` ***heißen.« Die Programmiersprache Java verwendet eine solche Regel. Eine andere Regel würde keinen Unterschied machen, solange wie der Compiler den Konstruktor von anderen Elementfunktionen unterscheiden kann.***

Mit einem Konstruktor sieht die Klasse `Student` wie folgt aus:

```
class Student
{
  public:
    // Datenelemente
    int   nSemesterHours;
    float dAverage;

    // Elementfunktionen
    Student()
    {
        nSemesterHours = 0;
        dAverage = 0.0;
    }
};

void fn()
{
    Student s; // erzeuge und initialisiere Objekt
    // ... Fortsetzung der Funktion ...
}
```

An der Stelle, an der `s` deklariert wird, fügt der Compiler einen Aufruf des Konstruktors `Student::Student( )` ein.

Dieser einfache Konstruktor wurde als Inline-Elementfunktion geschrieben. Konstruktoren können auch als Outline-Funktionen geschrieben werden, z.B.:

```
class Student
{
  public:
    // Datenelemente
    int   nSemesterHours;
    float dAverage;

    // Elementfunktionen
    Student();
};

Student::Student()
{
    nSemesterHours = 0;
    dAverage = 0.0;
}
int main(int nArgc, char* pszArgs)
{
```

```
    Student s; // erzeuge und initialisiere Objekt
    return 0;
}
```

Ich habe eine kleine Funktion `main( )` hinzugefügt, damit Sie das Programm ausführen können. Sie sollten mit Ihrem Debugger in Einzelschritten durch das Programm gehen, bevor sie hier fortfahren.

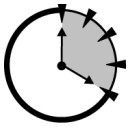
Wenn Sie in Einzelschritten durch dieses Beispielprogramm geben, kommt die Kontrolle schließlich zur Deklaration `Student s;`. Führen Sie Step In einmal aus, und die Kontrolle springt magischerweise nach `Student::Student( )`. Gehen Sie in Einzelschritten durch den Konstruktor. Wenn die Funktion fertig ist, kehrt die Kontrolle zur Anweisung nach der Deklaration zurück.

Mehrere Objekte können in einer Zeile deklariert werden. Gehen Sie nochmals in Einzelschritten durch die Funktion `main( )`, die wie folgt deklariert ist:

```
int main(int nArgc, char* pszArgs)
{
    Student s[5];    // erzeuge Array von Objekten
}
```

Der Konstruktor wird fünfmal aufgerufen, einmal für jedes Element des Array.

Wenn Sie den Debugger nicht zum Laufen bringen (oder sich damit einfach nicht beschäftigen möchten), fügen Sie eine Ausgabeanweisung in den Konstruktor ein, damit Sie auf dem Bildschirm sehen können, wenn der Konstruktor aufgerufen wird. Der Effekt ist nicht so dramatisch, aber überzeugend.



20 Min.

Grenzen des Konstruktors

Der Konstruktor kann nur automatisch aufgerufen werden. Er kann nicht wie eine normale Elementfunktion aufgerufen werden. D.h. Sie können ihn nicht so wie in folgendem Beispiel verwenden, um ein `Student`-Objekt neu zu initialisieren:

```
void fn()
{
    Student s; // erzeuge und initialisiere Objekt
    // ... weitere Anweisungen ...
    s.Student(); // initialisiere Objekt erneut -
                 // das funktioniert nicht
}
```

Der Konstruktor hat keinen Rückgabewert, noch nicht einmal `void`. Die Konstruktoren, die Sie hier sehen, haben alle eine `void`-Argumentliste.

Der Konstruktor ohne Argumente wird auch als Defaultkonstruktor bezeichnet.

Der Konstruktor kann andere Funktionen aufrufen. Wenn wir ein Objekt neu initialisieren wollen, schreiben wir das Folgende:

```
class Student
{
  public:
    // Datenelemente
    int   nSemesterHours;
    float dAverage;

    // Elementfunktionen
    // Konstruktor - initialisiert das Objekt bei
    //               Erzeugung automatisch
    Student()
    {
        init();
    }

    // init - initialisiere das Objekt
    void init()
    {
        nSemesterHours = 0;
        dAverage = 0.0;
    }
};

void fn()
{
    Student s; // erzeuge und initialisiere Objekt

    // ... weitere Anweisungen ...
    s.init();  // initialisiere Objekt erneut
}
```

Hierbei ruft der Konstruktor eine allgemein verfügbare Funktion `init( )` auf, die die Initialisierung durchführt.

Konstruktion von Datenelementen

Die Datenelemente eines Objektes werden zur gleichen Zeit wie das Objekt erzeugt. Die Datenelemente werden tatsächlich in der Reihenfolge erzeugt, in der sie im Quelltext stehen, und vor dem Rest der Klasse. Betrachen Sie das Programm ConstructMember in Listing 19-1. Ausgabeanweisungen wurden in die Konstruktoren der einzelnen Klassen eingefügt, damit Sie sehen können, in welcher Reihenfolge die Objekte erzeugt werden.

Listing 19-1: Das Programm ConstructMembers

```
// ConstructMembers - erzeuge ein Objekt, das
//                    Datenelemente hat, die selber
//                    Objekte einer Klasse sind
#include <stdio.h>
#include <iostream.h>
class Student
{
  public:
    Student()
    {
        cout << »Konstruktor Student\n«;
    }
};

class Teacher
{
  public:
    Teacher()
    {
        cout << »Konstruktor Teacher\n«;
    }
};

class TutorPair
{
  public:
    Student student;
    Teacher teacher;
    int     noMeetings;

    TutorPair()
    {
        cout << »Konstruktor TutorPair \n«;
        noMeetings = 0;
    }
};

int main(int nArgc, char* pArgs[])
{
    cout << »Erzeuge ein TutorPair\n«;

    TutorPair tp;

    cout << »Zurück in main\n«;
    return 0;
}
```

Ausführen des Programms erzeugt die folgende Ausgabe:

```
Erzeuge ein TutorPair
Konstruktor Student
Konstruktor Teacher
Konstruktor TutorPair
Zurück in main
```

Bei der Erzeugung von `tp` in `main( )` wird der Konstruktor von `TutorPair` automatisch aufgerufen. Bevor die Kontrolle an den Body des Konstruktors `TutorPair` übergeht, werden die Konstruktoren der beiden Elementobjekte – `student` und `teacher` – aufgerufen.

Der Konstruktor von `Student` wird zuerst aufgerufen, weil das Element `student` zuerst deklariert ist. Dann wird der Konstruktor von `Teacher` aufgerufen. Nachdem die Objekte erzeugt sind, geht die Kontrolle auf die öffnende Klammer über, und der Konstruktor von `TutorPair` darf den Rest des Objektes initialisieren.

Es würde nicht gehen,* `TutorPair` *für die Initialisierung von* `student` *und* `teacher` *verantwortlich zu machen. Jede Klasse ist für die Initialisierung ihrer Objekte selber zuständig.

19.1.2 Der Destruktor

So wie Objekte erzeugt werden, werden sie auch wieder vernichtet. Wenn eine Klasse einen Konstruktor zur Initialisierung hat, sollte sie auch eine spezielle Elementfunktion besitzen, die als Destruktor bezeichnet wird, um ein Objekt zu vernichten.

Der *Destruktor* ist eine spezielle Elementfunktion, die aufgerufen wird, wenn ein Objekt vernichtet wird, oder in C++-Sprechweise »destruiert« wird.

Eine Klasse kann in einem Konstruktor Ressourcen anlegen; diese Ressourcen müssen wieder freigegeben werden, bevor das Objekt aufhört zu existieren. Wenn der Konstruktor z.B. eine Datei öffnet, muss diese Datei wieder geschlossen werden. Oder wenn der Konstruktor Speicher vom Heap alloziert, muss dieser Speicher wieder freigegeben werden, bevor das Objekt verschwindet. Der Destruktor erlaubt es der Klasse, dieses Aufräumen automatisch durchzuführen, ohne sich auf die Anwendung verlassen zu müssen, dass die entsprechende Elementfunktion aufgerufen wird.

Das Destruktorelement hat den gleichen Namen wie die Klasse, mit einem vorangestellten Tildezeichen (~). Wie ein Konstruktor hat auch der Destruktor keinen Rückgabetyp. Z.B. sieht die Klasse `Student` mit einem hinzugefügten Destruktor wie folgt aus:

```
class Student
{
  public:
    // Datenelemente
    int   nSemesterHours;
    float dAverage;

    // ein Array für die einzelnen Grade
    int*  pnGrades;

    // Elementfunktionen
```

```
    // Konstruktor - aufgerufen bei Objekterzeugung;
    //              initialisiert die Elemente - die

    //              Allozierung eines Array vom
    //              Heap eingeschlossen
    Student()
    {
        nSemesterHours = 0;
        dAverage = 0.0;

        // Alloziere Platz für 50 Grade
        pnGrades = new int[50];
    }

    // Destruktor - aufgerufen bei Vernichtung des
    //             Objektes, um den Heapspeicher
    //             zurückzugeben
    ~Student()
    {
        // gib den Speicher an den Heap zurück
        delete pnGrades;
        pnGrades = 0;
    }
};
```

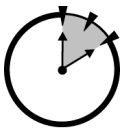
Wenn mehr als ein Objekt vernichtet wird, werden die Destruktoren in der umgekehrten Reihenfolge der Konstruktoren aufgerufen. Das Gleiche gilt bei der Vernichtung von Objekte, die Klassenobjekte als Datenelemente enthalten. Listing 19-2 zeigt die Ausgabe des Programms aus Listing 19-1 mit hinzugefügten Destruktoren in den drei Klassen:

Listing 19-2: Ausgabe von ConstructMembers, nachdem Destruktor eingefügt wurde

```
Erzeuge ein TutorPair
Konstruktor Student
Konstruktor Teacher
Konstruktor TutorPair
Zurück in main
Destruktor TutorPair
Destruktor Teacher
Destruktor Student
```

Das gesamte Programm ist auf der beiliegenden CD-ROM enthalten.

Der Konstruktor von `TutorPair` wird bei der Deklaration von tp aufgerufen. Die `Student`- und `Teacher`-Datenobjekte werden in der Reihenfolge erzeugt, wie sie in `TutorPair` enthalten sind, bevor die Kontrolle an den Body von `TutorPair( )` übergeben wird. Wenn die schließende Klammer von `main( )` erreicht wird, verlässt `tp` seinen Gültigkeitsbereich. C++ ruft `~TutorPair( )` auf, um `tp` zu vernichten. Nachdem der Destruktor das `TutorPair`-Objekt vernichtet hat werden die Destruktoren `~Student( )` und `~Teacher( )` aufgerufen, um die Datenelemente zu vernichten.



10 Min.

19.2 Zugriffskontrolle

Ein Objekt mit einem bekannten Zustand zu initialisieren, ist nur die halbe Miete. Die andere Hälfte besteht darin, sicherzustellen, dass externe Funktionen nicht in das Objekt »eindringen« können und Unsinn mit seinen Datenelementen anstellen können.

Externen Funktionen den Zugriff auf die Datenelemente zu erlauben, ist das Gleiche, wie den Zugriff auf die Interna meiner Mikrowelle zu gestatten. Wenn ich die Mikrowelle aufmache, und die Verkabelung ändere, kann ich den Entwickler der Mikrowelle nur schwerlich für die Folgen verantwortlich machen.

19.2.1 Das Schlüsselwort protected

C++ erlaubt es Klassen auch, Datenelemente zu deklarieren, die für Nichtelementfunktionen nicht zugreifbar sind. C++ verwendet das Schlüsselwort `protected`, um Klassenelemente so zu markieren, dass Sie für externe Funktionen zugreifbar sind.

Ein Klassenelement ist protected, wenn es nur von anderen Elementen der Klasse angesprochen werden kann.

Der Gegensatz zu protected ist public. Ein public-Element kann von Element- und Nichtelementfunktionen angesprochen werden.

Z.B. in der folgenden Version von `Student` sind nur die Funktionen `grade(double, int)` und `grade( )` für externe Funktionen zugreifbar.

```
// ProtectedMembers - demonstriert die Verwendung von
//                    protected-Elementen
#include <stdio.h>
#include <iostream.h>

// Student
class Student
{
  protected:
```

```
    double dCombinedScore;
    int    nSemesterHours;

  public:
    Student()
    {
       dCombinedScore = 0;
       nSemesterHours = 0;
    }

    // grade - addiere einen weiteren Grad
    double grade(double dNewGrade, int nHours)
    {
        // wenn die Werte gültig sind ...
        if (dNewGrade >= 0 && dNewGrade <= 4.0)
        {
            if (nHours >0 && nHours <= 5)
            {
                // ...führe Update durch
                dCombinedScore += dNewGrade * nHours;
                nSemesterHours += nHours;
            }
        }
        return grade();
    }

    // grade - gib den Grad zurück
    double grade()
    {
        return dCombinedScore / nSemesterHours;
    }

    // semesterHours - gib die aktuelle Anzahl der
    //                 Semesterstunden zurück
    int semesterHours()
    {
        return nSemesterHours;
    }
};

int main(int nArgc, char* pszArgs[])
{
    // erzeuge ein Student-Objekt vom Heap
    Student* pS = new Student;

    // addiere ein paar Grade
    pS->grade(2.5, 3);
    pS->grade(4.0, 3);
    pS->grade(3, 3);

    // hole den aktuellen Grad
    cout << »Der Grad ist » << pS->grade() << »\n«;

    return 0;
}
```

Diese Version von `Student` hat zwei Datenelemente. `dCombinedScore` zeigt die Summe der gewichteten Grade, während `nSemesterHours` die Gesamtsumme der Semesterstunden widerspiegelt. Die Funktion `grade(double, int)` führt ein Update für die Summe der gewichteten Grade und die Anzahl der Semesterstunden aus. Die Funktion `grade( )` gibt, ihrem Namen entsprechend, den derzeitigen Grad zurück, den sie als Verhältnis der mittleren Grade und der Gesamtanzahl der Semesterstunden berechnet.

`grade(double, int)` ***addiert den Effekt eines neuen Kurses zum gesamten Grad, während*** `grade(void)` ***den aktuellen Grad zurückgibt. Diese Zweiteilung, bei der eine Funktion einen Wert updatet, während die andere nur den Wert zurückgibt, ist häufig anzutreffen.***

Eine Funktion `grade( )`, die den Wert eines Datenelementes zurückgibt, wird als Zugriffsfunktion bezeichnet, weil sie einen kontrollierten Zugriff auf ein Datenelement bereitstellt.

Obwohl die Funktion `grade(double, int)` nicht fehlersicher ist, zeigt Sie doch ein wenig, wie eine Klasse sich selber schützen kann. Die Funktion führt eine Reihe rudimentärer Tests aus, um sicherzustellen, dass die übergebenen Daten Sinn machen. Die Klasse Student weiß, dass gültige Grade zwischen 0 und 4 liegen. Außerdem weiß die Klasse, dass die Anzahl der Semesterstunden eines Kurses zwischen 0 und 5 liegen (die Obergrenze ist meine eigene Erfindung).

Die elementaren Überprüfungen, die von der Methode `grade( )` durchgeführt werden, stellen eine gewisse Integrität der Daten sicher, vorausgesetzt, die Datenelemente sind für externe Funktionen nicht zugreifbar.

Es gibt noch einen weiteren Kontroll-Level, der als private bezeichnet wird. Der Unterschied von protected und private wird deutlich, wenn wir in Sitzung 32 die Vererbung diskutieren.

Die Elementfunktion `semesterHours( )` tut nicht mehr, als den Wert von `nSemesterHours` zurückzugeben.

Eine Funktion, die nichts anderes tut, als externen Funktionen Zugriff auf die Werte von Datenelementen zu geben, wird *Zugriffsfunktion* genannt. Eine Zugriffsfunktion erlaubt es Nichtelementfunktionen, den Wert eines Datenelementes zu lesen, ohne es verändern zu können.

Eine Funktion, die Zugriff auf die protected-Elemente einer Klasse hat, wird als *vertrauenswürdige* Funktion bezeichnet. Alle Elementfunktionen sind vertrauenswürdig. Auch Nichtelementfunktionen können als vertrauenswürdig bezeichnet werden durch die Verwendung des Schlüsselwortes `friend` (Freund). Eine Funktion, die als Freund einer Klasse bezeichnet ist, ist vertrauenswürdig. Alle Elementfunktionen einer `friend`-Klasse sind Freunde. Der richtige Gebrauch von `friend` geht über die Zielsetzung dieses Buches hinaus.

19.2.2 Statische Datenelemente

Unabhängig davon, wie viele Elemente protected sind, ist unsere Klasse `LinkList` aus Sitzung 15 doch noch verwundbar durch externe Funktionen durch den globalen Kopfzeiger. Was wir eigentlich möchten, ist diesen Zeiger unter den Schutz der Klasse zu stellen. Wie können jedoch kein normales Datenelement verwenden, weil diese für jede Instanz von `LinkList` separat angelegt werden – es kann nur einen Kopfzeiger in einer verketteten Liste geben. C++ bietet hierfür eine Lösung mit statischen Datenelementen.

Ein *statisches Datenelement* wird nicht für alle Objekte der Klasse separat instanziiert. Alle Objekte der Klasse teilen sich das gleiche statische Element.

Die Syntax zur Deklaration statischer Datenelemente ist ein bißchen unangenehm:

```
class LinkedList
{
  protected:
    // deklariere pHead als Element der Klasse,
    // aber gleich für alle Objekte
    static LinkedList* pHead;

    // der Zeiger pNext wird für jedes Objekt
    // separat angelegt
    LinkedList* pNext;

    // addHead - fügt ein Datenelement an den Anfang
    //           der Liste an
    void addHead()
    {
        // verkette den aktuellen Eintrag und
        // den Kopf der Liste
        pNext = pHead;

        // setze den Kopfzeiger auf das aktuelle
        // Objekt (this)
        pHead = this;
    }

    // ... der Rest des Programms ...
};

// alloziere jetzt einen Speicherbereich, in dem die
// statischen Elemente abgelegt werden können;
// stellen Sie sicher, dass die Objekte hier
// initialisiert werden, weil der Konstruktor
// das nicht tut
LinkedList* LinkedList::pHead = 0;
```

Die statische Deklaration in der Klasse macht `pHead` zu einem Element, alloziert aber keinen Speicher dafür. Dies muss außerhalb der Klasse geschehen, wie oben zu sehen ist.

Die gleiche Funktion `addHead( )` greift auf `pHead` zu, wie sie auf jedes andere Datenelement zugreifen würde. Zuerst lässt sie den `pNext`-Zeiger des aktuellen Objekts auf den Anfang der Liste zeigen – der Eintrag, auf den `pHead` zeigt. Danach verändert es diesen Zeiger, auf den aktuellen Eintrag zu zeigen.

Erinnern Sie sich daran, dass die Adresse des aktuellen Eintrags über das Schlüsselwort `this` angesprochen werden kann.

So einfach `addHead( )` ***ist, untersuchen Sie die Funktion dennoch genau: Alle Objekte der Klasse*** `LinkedList` ***haben das gleiche Element*** `pHead`***, jedes Objekt hat jedoch seinen eigenen*** `pNext`***-Zeiger.***

Es ist auch möglich, eine Elementfunktion statisch zu deklarieren; in diesem Buch verwenden wir jedoch keine solchen Funktionen.

Zusammenfassung

Der Konstruktor ist eine spezielle Elementfunktion, die C++ automatisch aufruft, wenn ein Objekt erzeugt wird, ob nun eine lokale Variable ihren Gültigkeitsbereich betritt oder ob ein Objekt vom Heap alloziert wird. Der Konstruktor ist verantwortlich dafür, die Datenelemente mit einem gültigen Zustand zu initialisieren. Die Datenelemente einer Klasse werden automatisch erzeugt, bevor der Konstruktor der Klasse aufgerufen wird. C++ ruft eine spezielle Funktion auf, wenn ein Objekt vernichtet wird, die als Destruktor bezeichnet wird.

- Der Klassenkonstruktor gibt der Klasse die Kontrolle darüber, wie Objekte erzeugt werden. Das verhindert, dass Objekte ihr Leben in einem illegalen Zustand beginnen. Konstruktoren werden wie die anderen Elementfunktionen deklariert, außer dass sie den Namen der Klasse tragen und keinen Rückgabetyp besitzen (nicht einmal `void`).
- Der Klassendestruktor gibt der Klasse die Chance, Ressourcen, die von einem Konstruktor belegt wurden, wieder freizugeben. Die häufigste Ressource ist Speicher.
- Eine Funktion protected zu deklarieren, macht sie nicht zugreifbar für nicht vertrauenswürdige Funktionen. Elementfunktionen werden automatisch als vertrauenwürdig angesehen.

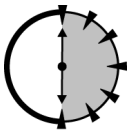
Selbsttest

1. Was ist ein Konstruktor? (Siehe »Der Konstruktor«)
2. Was ist falsch daran, eine Funktion `init( )` zur Initialisierung eines Objektes aufzurufen, wenn es angelegt wird? (Siehe »Der Konstruktor«)
3. Was ist der vollständige Name eines Konstruktors der Klasse `Teacher`? Was ist der Rückgabetyp? (Siehe »Der Konstruktor«)
4. In welcher Reihenfolge werden Datenelemente eines Objektes angelegt? (Siehe »Konstruktion von Datenelementen«)
5. Was ist der vollständige Name des Destruktors der Klasse `Teacher`? (Siehe »Der Destruktor«)
6. Was ist die Bedeutung des Schlüsselwortes `static`, wenn es in Verbindung mit Datenelementen verwendet wird? (Siehe »Statische Datenelemente«)

Klassenkonstruktoren II

Checkliste

- ☑ Konstruktoren mit Argumenten erzeugen
- ☑ Argumente an die Konstruktoren von Datenelementen übergeben
- ☑ Konstruktionsreihenfolge der Datenelemente bestimmen
- ☑ Spezielle Eigenschaften des Kopierkonstruktors bestimmen



30 Min.

Ein sehr einfacher Konstruktor arbeitet gut in der einfachen Klasse `Student` in Sitzung 19. Ohne dass die Klasse `Student` sehr komplex wird, treten die Begrenzungen des Konstruktors nicht zu Tage. Bedenken Sie z.B., dass ein `Student` einen Namen und eine Sozialversicherungsnummer hat. Der Konstruktor aus Sitzung 19 hat keine Argumente, so hat er keine andere Wahl, als das Objekt »leer« zu initialisieren. Der Konstruktor soll ein gültiges Objekt erzeugen – ein namenloser Student ohne Sozialversicherungsnummer stellt sicherlich keinen gültigen Studenten dar.

20.1 Konstruktoren mit Argumenten

C++ erlaubt es dem Programmierer, Konstruktoren mit Argumenten zu definieren, wie in Listing 20-1 zu sehen ist.

Listing 20-1: Definition eines Konstruktors mit Argumenten

```
// NamedStudent - zeigt, wie Argumente im Konstruktor
//                eine Klasse realistischer machen
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Student
class Student
{
  public:
```

```
    Student(char *pszName, int nID)
    {
        // Speichere eine Kopie des Namens:
        // alloziere vom Heap die gleiche Größe
        // wie die übergebene Zeichenkette
        int nLength = strlen(pszName) + 1;
        this->pszName = new char[nLength];

        // kopiere den übergebenen Namen in
        // den neu allozierten Puffer
        strcpy(this->pszName, pszName);

        // speichere die Studenten-ID
        this->nID = nID;
    }

    ~Student()
    {
        delete pszName;
        pszName = 0;
    }

  protected:
    char* pszName;
    int   nID;
};

void fn()
{
    // erzeuge einen lokalen Studenten
    Student  s1(»Stephen Davis«, 12345);

    // erzeuge einen Studenten vom Heap
    Student* pS2 = new Student(»Ron Davis«, 67890);

    // stelle sicher, dass aller Speicher an den
    // Heap zurückgegeben wird
    delete pS2;
}
```


Der Konstruktor `Student(char*, int)` beginnt mit der Allozierung eines Speicherblocks vom Heap, um eine Kopie der übergebenen Zeichenkette zu speichern. Die Funktion `strlen( )` gibt die Anzahl der Bytes in der Zeichenkette zurück; weil `strlen( )` das *Nullzeichen* nicht mitzählt, müssen wir 1 addieren, um die Anzahl Bytes zu erhalten, die wir vom Heap benötigen. Der Konstruktor `Student( )` kopiert die übergebene Zeichenkette in diesen Speicherblock. Schließlich weist der Konstruktor den Studenten die ID zu.

Die Funktion `fn( )` erzeugt zwei `Student`-Objekte, eins lokal in der Funktion und eins vom Heap. In beiden Fällen übergibt `fn( )` den Namen und die ID an das `Student`-Objekt, das erzeugt wird.

In diesem Beispiel `NamedStudent` ***tragen die Argumente des Konstruktors*** `Student` ***die gleichen Namen wie die Datenelemente, die sie initialisieren. Das ist keine Anforderung von C++, sondern nur eine Konvention, die ich bevorzuge. Eine lokal definierte Variable hat jedoch Vorrang vor einem Datenelement mit dem gleichen Namen. Somit bezieht sich eine unqualifizierte Referenz innerhalb von*** `Student( )` ***auf*** `pszName` ***auf das Argument;*** `this->pszName` ***gestattet jedoch immer, das Datenelement anzusprechen, unabhängig von allen lokal deklarierten Variablen. Während einige diese Praxis verwirrend finden, denke ich, dass sie die Verbindung zwischen den Argumenten und den Datenelementen unterstreicht. In jedem Fall sollten Sie mit dieser üblichen Praxis vertraut sein.***

Konstruktoren können überladen werden, wie Funktionen mit Argumenten überladen werden können.

Erinnern Sie sich, dass das Überladen einer Funktion bedeutet, dass Sie zwei Funktionen mit dem gleichen Namen haben, die sich durch ihre unterschiedlichen Argumenttypen unterscheiden.

C++ wählt den entsprechenden Konstruktor, basierend auf den Argumenten in der Deklaration. Z.B. kann die Klasse `Student` gleichzeitig drei Konstruktoren haben, wie im folgenden Schnipsel:

```
#include <iostream.h>
#include <string.h>
class Student
{
  public:
    // die verfügbaren Konstruktoren
    Student();
    Student(char* pszName);
    Student(char* pszName, int nID);
    ~Student();

  protected:
    char* pszName;
    int   nID;
};

// das Folgende ruft jeden Konstruktor auf
int main(int nArgs, char* pszArgs[])
{
    Student noName;
    Student freshMan(»Smel E. Fish«);
    Student xfer(»Upp R. Classman«, 1234);
    return 0;
}
```

Weil das Objekt `noName` ohne Argument erscheint, wird es von dem Konstruktor `Student::Student( )` erzeugt. Dieser Konstruktor wird als *Defaultkonstruktor* (oder auch *void-Konstruktor*) bezeichnet. (Ich persönlich bevorzuge den zweiten Begriff, aber der erste ist üblicher, weshalb er in diesem Buch verwendet wird.)

Es ist oft der Fall, dass der einzige Unterschied zwischen einem Konstruktor und einem anderen das Hinzufügen eines Defaultwertes für ein fehlendes Argument ist. Nehmen Sie z.B. an, dass ein Student, der ohne ID erzeugt wird, die ID 0 erhält. Um Extraarbeit zu vermeiden, ermöglicht es C++ dem Programmierer, einen Defaultwert zuzuweisen. Ich hätte die beiden letzten Konstruktoren wie folgt kombinieren können:

```
Student(char* pszName, int nID = 0);

Student s1(»Lynn Anderson«, 1234);
Student s2(»Stella Prater«);
```

Beide, `s1` und `s2`, werden durch den gleichen Konstruktor `Student(char*, in)` konstruiert. Im Falle von Stella, wird ein Defaultwert von 0 dem Konstruktor bereitgestellt.

Defaultwerte für Argumente können in jeder Funktion angegeben werden; ihren größten Nutzen finden Sie jedoch bei der Reduzierung der Konstruktorenanzahl.

20.2 Konstruktion von Klassenelementen

C++ konstruiert Datenelementobjekte zur gleichen Zeit, wie das Objekt selber. Es gibt keinen offensichtlichen Weg, Argumente bei der Konstruktion von Datenelementen zu übergeben.

In den Beispielen in Sitzung 19 gab es keinen Grund, Argumente an die Konstruktoren der Datenelemente zu übergeben – diese Version von `Student` hat sich auf die Anwendung verlassen, die Funktion `init( )` aufzurufen, um Objekte mit gültigen Werten zu initialisieren.

Betrachten Sie Listing 20-2, in dem die Klasse `Student` ein Objekt der Klasse `StudentId` enthält. Ich habe beide mit Ausgabeanweisungen in den Konstruktoren ausgestattet, um zu sehen, was passiert.

Listing 20-2: Konstruktor Student mit einem Elementobjekt aus der Klasse StundentId

```
// DefaultStudentId - erzeuge ein Student-Objekt mit
//                    der nächsten verfügbaren ID
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// StudentId
int nNextStudentId = 0;
class StudentId
```

```
{
  public:
    StudentId()
    {
        nId = ++nNextStudentId;
        cout << »Konstruktor StudentId » << nId << »\n«;
    }

    ~StudentId()
    {
        cout << »Destruktor StudentId »
             << nId << »\n«;
  protected:
    int nId;
};

// Student
class Student
{
  public:
    // Constructor - erzeuge Student mit
    //              dem gegebenen Namen
    //              (»kein Name« als Default)
    Student(char *pszName  = »kein Name«)
    {
       cout << »Konstruktor Student » << pszName << »\n«;

       // kopiere übergebene Zeichenkette in Element
       this->pszName = new char[strlen(pszName) + 1];
       strcpy(this->pszName, pszName);
    }

    ~Student()
    {
        cout << »Destruktor Student » << pszName << »\n«;
        delete pszName;
        pszName = 0;
    }

  protected:
    char* pszName;
    StudentId id;
};

int main(int nArgs, char* pszArg[])
{
    Student s(»Randy«);
    return 0;
}
```

Eine `Studenten-ID` wird jedem Studenten zugeordnet, wenn ein `Student`-Objekt erzeugt wird. In diesem Beispiel werden IDs fortlaufend vergeben unter Verwendung der globalen Variable `next-StudentId`, die die nächste zu vergebende ID enthält.

Die Ausgabe dieses einfachen Programms sieht so aus:

```
Konstruktor StudentId 1
Konstruktor Student Randy
Destruktor Student Randy
Destruktor StudentId 1
```

Beachten Sie, dass die Meldung vom Konstruktor `StudentId( )` vor der Ausgabe des Konstruktors `Student( )` erscheint, und die Meldung des `StudentId`-Destruktors nach der Ausgabe des `Student`-Destruktors erscheint.

Wenn alle Konstruktoren hier etwas ausgeben, könnten sie denken, dass jeder Konstruktor etwas ausgeben muss. Die meisten Konstruktoren geben nichts aus. Konstruktoren in Büchern tun es, weil die Leser normalerweise den guten Rat des Autors ignorieren, Schritt für Schritt durch die Programme zu gehen.

Wenn der Programmierer keinen Konstruktor bereitstellt, ruft der Defaultkonstruktor, der von C++ automatisch bereitgestellt wird, die Defaultkonstruktoren der Datenelemente auf. Das Gleiche gilt für den Destruktor, der automatisch die Destruktoren der Datenelemente aufruft, die Destruktoren haben. Der Defaultdestruktor von C++ tut das Gleiche.

Das ist alles großartig für den Defaultkonstruktor. Aber was ist, wenn wir einen anderen Konstruktor als den Defaultkonstruktor aufrufen wollen? Wo tun wir das Objekt hin? Um das zu verdeutlichen, lassen Sie uns annehmen, dass, statt der Berechnung der `Studenten`-ID, diese als Argument des Konstruktors bereitgestellt wird.

Lassen Sie uns ansehen, wie es nicht funktioniert. Betrachten Sie das Programm in Listing 20-3.

Listing 20-3: So kann ein Datenelementobjekt nicht erzeugt werden

```
// FalseStudentId - es wird versucht, die StudentId
//                  mit einem Konstruktor zu
//                  erzeugen, der nicht default ist
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// StudentId
int nNextStudentId = 0;
class StudentId
{
  public:
    StudentId(int nId = 0)
    {
        this->nId = nId;
        cout << »Konstruktor StudentId » << nId << »\n«;
    }

    ~StudentId()
    {
        cout << »Destruktor StudentId »
             << nId << »\n«;
  protected:
```

```
    int nId;
};

// Student
class Student
{
  public:
    // Konstruktor - erzeuge Student mit
    //              dem gegebenen Namen und
    //              Studenten-ID
    Student(char *pszName  = »kein Name«,
            int ssId = 0)
    {
       cout << »Konstruktor Student » << pszName << »\n«;

       // kopiere übergebene Zeichenkette in Element
       this->pszName = new char[strlen(pszName) + 1];
       strcpy(this->pszName, pszName);

        // das funktioniert nicht
        StudentId id(ssId); // konstruiere Studenten-ID
    }

    ~Student()
    {
        cout << »Destruktor Student » << pszName << »\n«;
        delete pszName;
        pszName = 0;
    }

  protected:
    char* pszName;
    StudentId id;
};

int main(int nArgs, char* pszArg[])
{
    Student s(»Randy«, 1234);
    cout << »Nachricht von main\n«;
    return 0;
}
```

Der Konstruktor für `StudentId` wurde so verändert, dass er einen Wert von außen entgegennimmt (der Defaultwert ist notwendig, um das Beispiel kompilieren zu können aus Gründen, die bald klar werden). Innerhalb des Konstruktors von `Student` hat der Programmierer (das bin ich) (clever) versucht, ein Objekt `StudentId` mit Namen `id` zu erzeugen.

Die Ausgabe des Programms zeigt ein Problem:

```
Konstruktor StudentId 0
Konstruktor Student Randy
Konstruktor StudentId 1234
Destruktor StudentId 1234
Nachricht von main
Destruktor Student Randy
Destruktor StudentId 0
```

Zum einen scheint der Konstruktor zweimal aufgerufen zu werden, das erste Mal mit null, bevor der Konstruktor `Student` beginnt, und das zweite Mal mit dem erwarteten Wert 1234 vom Konstruktor `Student` aus. Offensichtlich wurde eine zweites Objekt `StudentId` erzeugt innerhalb des Konstruktors `Student`, das vom `StudentId`-Datenelement verschieden ist.

Die Erklärung für dieses bizarre Verhalten ist, dass das Datenelement `id` bereits existiert, wenn der Body des Konstruktors betreten wird. Anstatt ein existierendes Datenelement `id` zu erzeugen, erzeugt die Deklaration im Konstruktor ein lokales Objekt mit dem gleichen Namen. Dieses lokale Objekt wird vernichtet, wenn der Konstruktor wieder verlassen wird.

Wir brauchen einen anderen Mechanismus um anzuzeigen »konstruiere das existierende Element; erzeuge kein neues.« C++ definiert die neue Konstruktion wie folgt:

```
class Student
{
  public:
    Student(char* pszName  = »no name«, int ssId = 0)
        : id(ssId) // konstruiere Datenelement id mit
                   // dem angegebenen Wert
    {
       cout << »Konstruktor Student » << pszName << »\n«;
       this->pszName = new char[strlen(pszName) + 1];
       strcpy(this->pszName, pszName);
    }
    // ...
};
```

Beachten Sie insbesondere die erste Zeile des Konstruktors. Wir haben das vorher noch nicht gesehen. Der Doppelpunkt »:« bedeutet, dass Aufrufe von Konstruktoren der Datenelemente der aktuellen Klasse folgen. Für den C++-Compiler liest sich die Zeile so: »Konstruiere das Element `id` mit dem Argument `ssId` des `Student`-Konstruktors. Alle Datenelemente, die nicht so behandelt werden, werden mit ihrem Defaultkonstruktor konstruiert.«

Das gesamte Programm befindet sich auf der beiliegenden CD-ROM unter dem Namen StudentId.

Das neue Programm erzeugt das erwartete Ergebnis:

```
Konstruktor StudentId 1234
Konstruktor Student Randy
Nachricht von main
Destruktor Student Randy
Destruktor StudentId 1234
```

Das Objekt `s` wird in `main( )` erzeugt mit dem Studentennamen »Randy« und einer Studenten-ID von 1234. Dieses Objekt wird am Ende von `main( )` automatisch vernichtet. Die »:«-Syntax muss auch bei der Wertzuweisung an Elemente vom Typ `const` oder mit Referenztyp verwendet werden. Betrachten Sie die folgende dumme Klasse:

```
class SillyClass
{
  public:
    SillyClass(int& i) : nTen(10), refI(i)
    {
    }
  protected:
    const int nTen;
    int& refI;
};
```

Wenn das Programm mit der Ausführung des Bodys des Konstruktors beginnt, sind die beiden Elemente `nTen` und `refI` bereits angelegt. Diese Datenelemente müssen initialisiert sein, bevor die Kontrolle an den Body des Konstruktors übergeht.

Jedes Datenelement kann mit der »im-Voraus«-Syntax deklariert werden.

20.3 Reihenfolge der Konstruktion

Wenn es mehrere Objekte gibt, die alle einen Konstruktor besitzen, kümmert sich der Programmierer normalerweise nicht um die Reihenfolge, in der die Dinge erzeugt werden. Wenn einer oder mehrere der Konstruktoren Seiteneffekte haben, kann die Reihenfolge einen Unterschied machen.

Ein Seiteneffekt ist eine Zustandsänderung, die durch eine Funktion verursacht wird, die aber nicht zu den Argumenten oder dem zurückgegebenen Objekt gehört. Wenn z.B. eine Funktion einer globalen Variablen einen Wert zuweist, wäre das ein Seiteneffekt.

Die Regeln für die Reihenfolge der Konstruktion sind:

- Lokale und statische Objekte werden in der Reihenfolge erzeugt, in der ihre Deklarationen aufgerufen werden.
- Statische Objekte werden nur einmal erzeugt.
- Alle globalen Objekte werden vor `main( )` konstruiert.
- Globale Objekte werden in keiner bestimmten Reihenfolge angelegt.
- Elemente werden in der Reihenfolge erzeugt, in der sie in der Klasse deklariert sind.

20.3.1 Lokale Objekte werden in der Reihenfolge konstruiert

Lokale Objekte werden in der Reihenfolge konstruiert, in der das Programm ihre Deklaration antrifft. Normalerweise ist das die gleiche Reihenfolge, in der das Programm die Objekte antrifft, außer die Funktion springt um gewisse Deklarationen herum. (So nebenbei, Deklarationen zu überspringen ist eine üble Sache. Es verwirrt den Leser und den Compiler.)

20.3.2 Statische Objekte werden nur einmal angelegt

Statische Objekte sind lokalen Variablen ähnlich, nur dass Sie nur einmal angelegt werden. Das ist auch zu erwarten, weil sie ihren Wert von einem Funktionsaufruf zum nächsten behalten. Anders als bei C, wo statische Variablen initialisiert werden, wenn das Programm beginnt, muss C++ warten, bis die Kontrolle das erste Mal ihre Deklaration durchläuft, um dann die Konstruktion der Variablen auszuführen. Betrachten sie das folgende triviale Programm:

```
class DoNothing
{
  public:
    DoNothing(int initial)
    {
    }
};

void fn(int i)
{
    static DoNothing staticDN(1);
    DoNothing localDN(2);
}
```

Die Variable `staticDN` wird beim ersten Aufruf von `fn( )` konstruiert. Beim zweiten Aufruf von `fn( )` konstruiert das Programm nur `localDN`.

20.3.3 Alle globalen Variablen werden vor main( ) erzeugt

Alle globalen Variablen betreten ihren Gültigkeitsbereich unmittelbar bei Programmstart. Somit werden alle globalen Objekte konstruiert, bevor die Kontrolle an `main( )` übergeht.

20.3.4 Keine bestimmte Reihenfolge für globale Objekte

Die Reihenfolge der Konstruktion von lokalen Objekten herauszufinden, ist einfach. Durch den Kontrollfluss ist eine Reihenfolge gegeben. Bei globalen Objekten gibt es keinen solchen Fluss, der eine Reihenfolge vorgeben würde. Alle globalen Variablen betreten gleichzeitig ihren Geltungsbereich, erinnern Sie sich? Nun, Sie werden sicherlich anführen, dass der Compiler doch eine Datei von oben nach unten durcharbeiten könnte, um alle globalen Variablen zu finden. Das ist für eine einzelne Datei richtig (und ich nehme an, dass die meisten Compiler genau das tun).

Unglücklicherweise bestehen die meisten Programme in der realen Welt aus mehreren Dateien, die getrennt kompiliert werden, und dann gemeinsam zum Programm zusammengefügt werden. Weil der Compiler keinen Einfluss darauf hat, wie diese Teile zusammengefügt werden, hat er keinen Einfluss darauf, wie globale Objekte von Datei zu Datei erzeugt werden.

Meistens ist die Reihenfolge gleichgültig. Hin und wieder können aber dadurch Bugs entstehen, die sehr schwierig zu finden sind. (Es passiert jedenfalls oft genug, um in einem Buch Erwähnung zu finden.)

Betrachten Sie das Beispiel:

```
class Student
{
  public:
    Student (unsigned id) : studentId(id)
    // unsigned ist vorzeichenloses int
    {
    }
    const unsigned studentId;
};

class Tutor
{
  public:
    // Konstruktor - weise dem Tutor einen Studenten
    //             zu durch Auslesen seiner ID
    Tutor(Student &s)
    {
        tutoredId = s.studentId;
    }
  protected:
    unsigned tutoredId;
};

// erzeuge global einen Student
Student randy(1234);

// weise diesem Student einen Tutor zu
Tutor   jenny(randy);
```

Hier weist der Konstruktor von `Student` eine Studenten-ID zu. Der Konstruktor von `Tutor` kopiert sich diese ID. Das Programm deklariert einen Studenten `randy` und weist dann diesem `Student` den Tutor `jenny` zu.

Das Problem ist, dass wir implizit annehmen, dass `randy` vor `jenny` konstruiert wird. Nehmen wir an, es wäre anders herum. Dann würde `jenny` mit einem Speicherblock initialisiert, der noch nicht in ein `Student`-Objekt verwandelt wurde und daher »Müll« als Studenten-ID enthält.

Dieses Beispiel ist nicht besonders schwer zu durchschauen und mehr als nur ein wenig konstruiert. Trotzdem können Probleme mit globalen Objekten, die in keiner bestimmten Ordnung konstruiert werden, sehr subtil sein. Um diese Probleme zu vermeiden, sollten Sie es dem Konstruktor eines globalen Objektes nicht gestatten, sich auf den Inhalt eines anderen globalen Objektes zu verlassen.

20.3.5 Elemente werden in der Reihenfolge ihrer Deklaration konstruiert

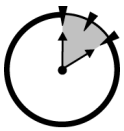
Elemente einer Klasse werden in der Reihenfolge konstruiert, in der sie in der Klasse deklariert wurden. Das ist nicht so offensichtlich wie es aussieht. Betrachen Sie das folgende Beispiel:

```
class Student
{
  public:
    Student (unsigned id, unsigned age) :
        sAge(age), sId(id)
    {
    }
    const unsigned sId;
    const unsigned sAge;
};
```

In diesem Beispiel wird `sId` konstruiert, bevor `sAge` überhaupt daran denken kann, als zweites in der Initialisierungsliste des Konstruktors zu stehen. Sie würden nur dann einen Unterschied in der Reihenfolge der Konstruktion bemerken, wenn beide Elemente Objekte wären, deren Konstruktoren gegenseitige Seiteneffekte haben.

20.3.6 Destruktoren in umgekehrter Reihenfolge wie Konstruktoren

Schließlich werden die Destruktoren, egal in welcher Reihenfolge die Konstruktoren aufgerufen wurden, in der umgekehrten Reihenfolge aufgerufen. (Es ist schön zu wissen, dass es in C++ wenigstens eine Regel ohne Wenn und Aber gibt.)



10 Min.

20.4 Der Kopierkonstruktor

Ein Kopierkonstruktor ist ein Konstruktor, der den Namen `X::X(&X)` trägt, wobei `X` ein Klassenname ist. D.h. er ist der Konstruktor einer Klasse `X`, und nimmt als Argument eine Referenz auf ein Objekt der Klasse `X`. Nun, ich weiß, dass das wirklich nutzlos klingt, aber denken Sie einen Augenblick darüber nach, was passiert, wenn Sie eine Funktion wie die folgende aufrufen:

```
void fn1(Student fs)
{
    // ...
}
int fn2()
{
    Student ms;
    fn1(ms);
    return 0;
}
```

Wie sie wissen, wird eine Kopie des Objektes `ms` – und nicht das Objekt selber – an die Funktion `fn1( )` übergeben. C++ könnte einfach eine exakte Kopie des Objektes machen und diese an `fn1( )` übergeben.

Das ist in C++ nicht akzeptabel. Erstens wird zur Konstruktion eines Objektes ein Konstruktor benötigt, selbst für das Kopieren eines existierenden Objektes. Zweitens, was ist, wenn wir keine einfache Kopie des Objektes haben möchten? (Lassen wir das »Warum?« für eine Weile.) Wir müssen in der Lage sein, anzugeben, wie die Kopie konstruiert werden soll.

Das Programm `CopyStudent` in Listing 20-4 demonstriert diesen Punkt.

Listing 20-4: Der Kopierkonstruktor in Aktion

```
// CopyStudent - demonstriert den Kopierkonstruktor
//               bei einer Wertübergabe
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class Student
{
  public:
    // Initialisierungsfunktion
    void init(char* pszName, int nId, char* pszPreamble = »\0«)
    {
        int nLength = strlen(pszName)
                    + strlen(pszPreamble)
                    + 1;
        this->pszName = new char[nLength];
        strcpy(this->pszName, pszPreamble);
        strcat(this->pszName, pszName);
        this->nId = nId;
    }

    // Konventioneller Konstruktor
    Student(char* pszName  = »noname«, int nId = 0)
    {
        cout << »Konstruktor Student » << pszName << »\n«;
        init(pszName, nId);
    }

    // Kopierkonstruktor
    Student(Student &s)
    {
        cout << »Kopierkonstruktor von »
             << s.pszName
             << »\n«;
        init(s.pszName, s.nId, »Kopie von »);
    }

    ~Student()
    {
       cout << »Destruktor » << pszName << »\n«;
       delete pszName;
    }
  protected:
    char* pszName;
    int   nId;
};
```

```
// fn - erhält Argument als Wert
void fn(Student s)
{
    cout << »In Funktion fn()\n«;
}

int main(int nArgs, char* pszArgs[])
{
    // erzeuge ein Student-Objekt
    Student randy(»Randy«, 1234);

    // übergib es als Wert an fn()
    cout << »Aufruf von fn()\n«;
    fn(randy);
    cout << »Rückkehr von fn()\n«;

    // fertig
    return 0;
}
```

Die Ausgabe des Programms sieht so aus:

```
Konstruktor Student Randy
Aufruf von fn( )
Kopierkonstruktor Student Randy
In Funktion fn( )
Destruktor Kopie von Randy
Rückkehr von fn( )
Destruktor Randy
```

Beginnend bei `main( )` sehen wir, wie dieses Programm arbeitet. Der normale Konstruktor erzeugt die erste Nachricht. `main( )` erzeugt die Nachricht »Aufruf von ...«. C++ ruft den Kopierkonstruktor auf, um eine Kopie von `randy` an `fn( )` zu übergeben, was die nächste Zeile der Ausgabe erzeugt. Die Kopie wird am Ende von `fn( )` vernichtet. Das originale Objekt `randy` wird am Ende von `main( )` vernichtet.

(Dieser Kopierkonstruktor macht ein wenig mehr, als nur eine Kopie des Objekts; er trägt die Phrase "`Kopie von`" am Anfang des Namens ein. Das war zu Ihrem Vorteil. Normalerweise sollten sich Kopierkonstruktoren darauf beschränken, einfach Kopien zu konstruieren. Aber prinzipiell können sie alles tun.)

20.4.1 Flache Kopie gegen tiefe Kopie

C++ sieht den Kopierkonstruktor als so wichtig an, dass C++ selber einen Kopierkonstruktor erzeugt, wenn Sie keinen definieren. Der Default-Kopierkonstruktor, den C++ bereitstellt, führt eine Element-zu-Element-Kopie durch.

Eine Element-zu-Element-Kopie durchzuführen, ist die offensichtliche Aufgabe eines Kopierkonstruktors. Außer zum Hinzufügen so unsinniger Dinge wie "`Kopie von`" an den Anfang eines Studentennamen, wo sonst würden wir eine Element-zu-Element-Kopie erstellen?

Betrachten Sie, was passiert, wenn der Konstruktor einen Speicherbereich vom Heap alloziert. Wenn der Kopierkonstruktor einfach eine Kopie davon macht, ohne seinen eigenen Speicher anzulegen, kommen wir in eine schwierige Situation: zwei Objekte denken, sie hätten exklusiven Zugriff

auf den gleichen Besitz. Das wird noch schlimmer, wenn der Destruktor für beide Objekte aufgerufen wird und sie beide versuchen, den Speicher zurückzugeben. Um das konkreter werden zu lassen, betrachten Sie die Klasse `Student` nochmals:

```
class Student
{
  public:
    Person(char *pszName)
    {
        pszName = new char[strlen(pszName) + 1];
        strcpy(pName, pN);
    }
    ~Person()
    {
        delete pszName;
    }
  protected:
    char* pszName;
};
```

Hier alloziert der Konstruktor Speicher vom Heap, um den Namen der Person zu speichern – der Destruktor gibt freundlicherweise den Speicher zurück, wie er soll. Wenn dieses Objekt als Wert an eine Funktion übergeben wird, würde C++ eine Kopie des `Student`-Objektes machen, den Zeiger `pszName` eingeschlossen. Das Programm hat dann zwei `Student`-Objekte, die auf den gleichen Speicherbereich zeigen, der den Namen des Studenten enthält. Wenn das erste Objekt vernichtet wird, wird der Speicherbereich zurückgegeben. Bei der Vernichtung des zweiten Objektes wird versucht, den gleichen Speicher erneut freizugeben – ein fataler Vorgang für ein Programm.

Speicher vom Heap ist nicht der einzige Besitz, der eine tiefe Kopie erforderlich macht, aber der häufigste. Offene Dateien, Ports und allozierte Hardware (wie z.B. Drucker) benötigen ebenfalls tiefe Kopien. Das sind dieselben Typen, die der Destruktor zurückgeben muss. Eine gute Daumenregel ist, dass ihre Klasse einen Kopierkonstruktor benötigt, wenn sie einen Destruktor benötigt, um Ressourcen wieder freizugeben.

20.4.2 Ein Fallback-Kopierkonstruktor

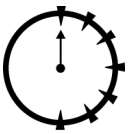
Es gibt Situationen, in denen Sie keine Kopien Ihrer Objekte erzeugt haben möchten. Das kann der Fall sein, weil Sie den Code, um eine tiefe Kopie zu erzeugen, nicht bereitstellen können oder wollen. Das ist auch der Fall, wenn das Objekt sehr groß ist und das Erzeugen einer Kopie, flach oder tief, einige Zeit benötigen würde.

Eine einfache Verteidigungsposition, um das Problem zu vermeiden, dass ohne Ihr Wissen flache Kopien von Objekten angelegt werden, ist das Erzeugen eines Kopierkonstruktors, der als protected deklariert ist.

Wenn Sie keinen Kopierkonstruktor selber erzeugen, legt C++ einen eigenen Kopierkonstruktor an.

Im folgenden Beispiel kann C++ keine Kopie der Klasse `BankAccount` anlegen, wodurch sichergestellt ist, dass das Programm nicht versehentlich eine Überweisung auf die Kopie eines Kontos und nicht auf das Konto selber ausführt.

```
class BankAccount
{
  protected:
    int nBalance;
    BankAccount(BankAccount& ba)
    {
    }

  public:
    // ... Rest der Klasse ...
};
```



0 Min.

Zusammenfassung

Der Konstruktor hat die Aufgabe, Objekte zu erzeugen und mit einem gültigen Zustand zu initialisieren. Viele Objekte haben jedoch keinen gültigen Default-Zustand. Z.B. ist es nicht möglich, ein gültiges `Student`-Objekt ohne einen Namen und eine ID zu erzeugen. Ein Konstruktor mit Argumenten ermöglicht es dem Programm, initiale Werte für neu erzeugte Objekte zu übergeben. Diese Argumente können auch an andere Konstruktoren der Datenelemente der Klasse weitergegeben werden. Die Reihenfolge, in der die Datenelemente konstruiert werden, ist im C++-Standard definiert (eines der wenigen Dinge, die wohldefiniert sind).

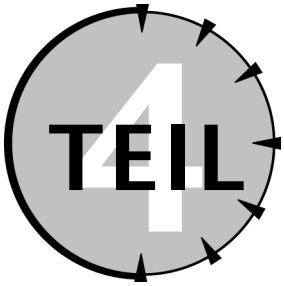
- Argumente von Konstruktoren erlauben die Festlegung eines initialen Zustandes von Objekten durch das Programm; für den Fall, dass diese initialen Werte nicht gültig sind, muss die Klasse einen Reservezustand haben.
- Es muss vorsichtig mit Konstruktoren umgegangen werden, die Seiteneffekte haben, wie z.B. das Ändern globaler Variablen, weil einer oder mehrere Konstruktoren kollidieren können, wenn sie ein Objekt erzeugen, das Datenelemente enthält, die selber Objekte sind.
- Ein spezieller Konstruktor, der Kopierkonstruktor, hat den Prototyp `X::X(&X)`, wobei `X` der Name der Klasse ist. Dieser Konstruktor wird dazu verwendet, eine Kopie eines existierenden Objektes zu erzeugen. Kopierkonstruktoren sind extrem wichtig, wenn das Objekt Ressourcen enthält, die im Destruktor an einen Ressourcen-Pool zurückgegeben werden.

Obwohl die Prinzipien der objektorientierten Programmierung erläutert wurden, sind wir noch nicht da, wo wir hin wollen. Wir haben ein Mikrowelle gebaut, wir haben eine Box um die arbeitenden Teile gebaut, und wir haben ein Benutzerinterface definiert; aber wir haben keine Beziehung zwischen Mikrowellen und anderen Ofentypen hergestellt. Das ist das Wesentliche der objektorientierten Programmierung, und wird in Teil 5 diskutiert.

Selbsttest

1. Warum sollte eine Klasse wie `Student` einen Konstruktor der Form `Student(char* pszName, int nId)` haben? (Siehe »Konstruktoren mit Argumenten«)
2. Wie können Sie etwas über die Reihenfolge aussagen, in der globale Objekte konstruiert werden? (Siehe »Keine bestimmte Reihenfolge für globale Objekte«)
3. Warum benötigt eine Klasse einen Kopierkonstruktor? (Siehe »Der Kopierkonstruktor«)
4. Was ist der Unterschied zwischen einer flachen Kopie und einer tiefen Kopie? (Siehe »Flache Kopie gegen tiefe Kopie«)

Samstagabend – Zusammenfassung



1. ***Denken Sie über den Inhalt Ihres Kleiderschrankes nach. Beschreiben Sie informell, was Sie dort finden.***

2. ***Betrachten Sie Schuhe. Beschreiben Sie das Interface von Schuhen.***

3. ***Benennen Sie zwei verschieden Typen von Schuhen. Was ist der Effekt, einen anstelle des anderen zu verwenden?***

4. ***Schreiben Sie einen Konstruktor für eine Klasse, die wie folgt definiert ist:***

```
class Link
{
    static Link* pHead;
    Link* pNextLink;
};
```

5. ***Schreiben Sie einen Kopierkonstruktor und einen Destruktor für die folgende Klasse*** `LinkedList`:

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class LinkedList
{
  protected:
    LinkedList* pNext;
    static LinkedList* pHead;
char* pszName;

  public:
    // Konstruktor - kopiere den Namen
    LinkedList(char* pszName)
    {
        int nLength = strlen(pszName) + 1;
        this->pszName = new char[nLength];
        strcpy(this->pszName, pszName);
        pNext = 0;
    }

    // diese Elementfunktionen soll es geben
    void addToList();
    void removeFromeList();
```

```
    // Destruktor -
    ~LinkedList()
    {
        // ... was kommt hier hin?
    }
    LinkedList(LinkedList& l)
    {
        // ... und hier?
    }
};
```

Hinweise:

a. Bedenken Sie, was passiert, wenn das Objekt immer noch in der verketteten Liste ist, nachdem es vernichtet wurde.

b. Erinnern Sie sich daran, dass Speicher, der vom Heap alloziert wurde, zurückgegeben werden sollte, bevor der Zeiger darauf verloren geht.