

Zeigs uns

# Zeiger auf Funktionen und Zeiger auf Zeiger

Mit Zeigern geht mehr, als bis jetzt gezeigt wurde, denn Sie können auch Zeiger auf Funktionen definieren.

THOMAS WÖLFER

Das Sie in C/C++ mit Zeigern ein prima Konstrukt zur Hand haben, ist im letzten Beitrag klar geworden: Ein sehr sinnvolles Einsatzgebiet – aber bei weitem nicht das einzige – ist die dynamische Speicherverwaltung. Mit Zeigern können Sie sich nicht nur Adressen von Daten merken, sondern auch die Adressen von Funktionen. Auch das ist sehr praktisch, denn damit können Sie Arrays aus Zeigern auf Funktionen bauen, und so etwas kann man immer gebrauchen: zum Beispiel, für ein einfaches Menüsystem.

Genau das wird in diesem Beitrag gebaut, und dazu wird das Programm aus dem vorhergehenden Beitrag erweitert. Bisher hat der FileViewer nur drei Funktionen: Man kann ihn beenden, man kann vorwärts scrollen, und man kann rückwärts scrollen. Die Anzahl der Funktionen soll an dieser Stelle nicht erweitert werden – stattdessen werden Sie eine Methode kennenlernen, um die Funktionalität des Programms besser zu verwalten. Man kann sich leicht vorstellen, dass der FileViewer mit einer ganzen Menge zusätzlicher Funktionen ausgestattet werden könn-

te. So wäre zum Beispiel eine Suchfunktion oder aber auch eine Funktion zum horizontalen Scrollen denkbar. (Apropos horizontales Scrollen: Haben Sie schon einmal ausprobiert, was beim jetzigen Viewer passiert, wenn eine Textzeile in der Textdatei länger als 80

```
004.cpp
// typedef fuer einfachere hinschreiben
// der function pointer
typedef void(*pFunc)(Node**);

// funktion zum vorwaerts blaettern
void Forward( Node** pNode)
{
    if( (*pNode)->pNext)
        (*pNode) = (*pNode)->pNext;
}

// funktion zum rueckwaerts blaettern
void Back( Node** pNode)
{
    if( (*pNode)->pPrev)
        (*pNode) = (*pNode)->pPrev;
}

// struktur fuer die menu-auswahl, bestehend
// aus der taste und der zugehoerenden
// funktion
struct MenuItem
{
    char    cKey; // taste
    pFunc   pF;  // funktion
};

// das menu
MenuItem arrMenu[] =
{
    'v', Forward,
    'r', Back
};
```

**ZEIGER AUF FUNKTIONEN** und Zeiger auf Zeiger sind schon deutlich schwieriger zu lesen. Allerdings brauchen Sie diese Funktionalität, um effiziente Programme schreiben zu können.

Zeichen wird? Hier bietet sich eine hervorragende Übungsmöglichkeit an, die Sie nicht ignorieren sollten.)

Wie auch immer: Bisher hat der Code, der auf die Befehlsauswahl reagiert in etwa folgenden Aufbau:

```
if( cSelect == "v")
{
```

```
} // hier passiert irgendwas
}

if( cSelect == "r")
{
    // hier passiert was anderes
}
```

Wenn Sie nun eine ganze Reihe von weiteren Funktionen in den Viewer einbauen, wird diese Methode, den Programmfluss zu steuern, irgendwann sehr unübersichtlich. Zwei Dinge müssen also unbedingt passieren:

- Der Code, der die Funktion implementiert muss in eine separate Funktion ausgelagert werden.
- Der Code, der entscheidet welche der Funktionen ausgeführt werden soll, muss kompakter werden.

Um die Sache kompakter und übersichtlicher zu machen, könnte man das weiter vorne im Heft vorgestellte *switch/case*-Statement verwenden und außerdem ein paar Symbole mit dem Präprozessor definieren. So würde der Code etwa den folgenden Aufbau erhalten:

```
// irgendwo weiter oben in der datei:
#define FUNC_VORWAERTS "v"
#define FUNC_RUECKWAERTS "r"

// funktionen zum scrollen
void Forwaerts()
{
    // tut irgend was
}

void Zurueck()
{
    // tut irgendwas
}

// dann in der schleife
switch( cSelect)
{
    case FUNC_VORWAERTS:
        Forwaerts();
        break;
    case FUNC_RUECKWAERTS:
        Rueckwaerts();
        break;
}
```



Dieser Ansatz hat seinen Liebreiz, allerdings auch seine Nachteile. Der größte Nachteil ist dabei der, dass Sie den Quellcode immer wieder bearbeiten müssen, wenn Sie eine neue Funktion in den Viewer einbauen. Das ist auch logisch: Jedesmal, wenn Sie eine neue Taste definieren, deren Betätigung dazu führen soll, dass eine bestimmte Funktion aufgerufen wird, müssen Sie das

genau muss für die Funktion hingeschrieben werden? Denn "FunktionDieDazuGehört" wird wohl kaum der richtige Ausdruck sein. Das ist eine gute Frage und wie so oft in C/C++ ist auch hier die Lösung die, dass Zeiger verwendet werden.

Sie haben bereits gelernt, dass der Name eines Character-Arrays mehr oder minder dem Zeiger auf das erste Element

theke weitergegeben. Das ist aber nicht alles, was Sie damit tun können – auch die C-Laufzeitbibliothek muss schließlich mit den Inhalten arbeiten, auf die die Zeiger zeigen.

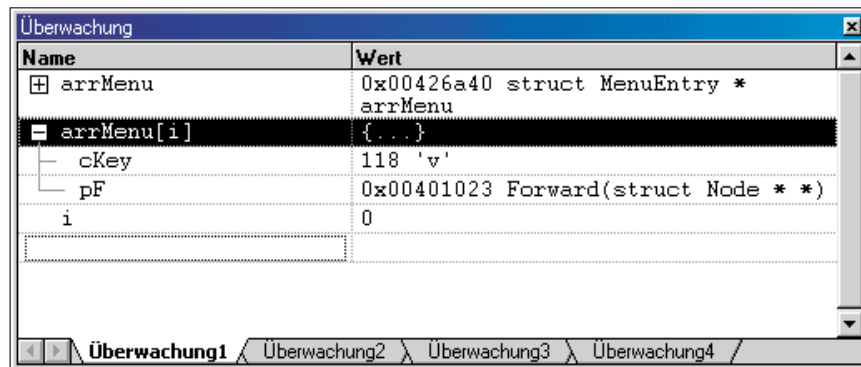
Angenommen, Sie haben einen Zeiger auf "char" und würden gerne das erste Zeichen davon ausgeben – was ist zu tun? Bisher kennen Sie die Methode `printf()`, der Sie einen solchen Zeiger übergeben können:

```
char* p = "Hallo Welt";  
printf( p );
```

Dieser Quellcode gibt den Text "Hallo Welt" aus. Was aber, wenn Sie nur das "H" ausgeben möchten? Dazu gibt es zum Beispiel die Funktion `putch()`, die als Parameter einen Char erwartet und diesen dann am Bildschirm ausgibt – stellt sich die Frage, wie Sie von dem "char\*" den Sie haben, auf einen echten "char" – also ohne "\*" – kommen. Das ist genau der Vorgang, der dereferenzieren genannt wird. Um den Inhalt eines Zeigers auszulesen, verwenden Sie ebenfalls einen "\*":

```
char* pString = "Hallo Welt";  
char c = *pString;  
putch( c );
```

Hier wird der Buchstabe "H" ausgegeben. Das geht dabei wie folgt vor sich: Sie haben einen Zeiger, der auf "char" zeigt, und zwar auf die Adresse des ersten Zeichens aus dem String "Hallo Welt". Den dereferenzieren Sie mit dem "\*" in der zweiten Zeile, und erhalten



**DAS ÜBERWACHUNGSFFENSTER DES DEBUGGERS IST IHR FREUND:** Ohne dessen Möglichkeiten würde die Fehlersuche in "verzeigerten" Strukturen deutlich schwieriger sein.

`switch`-Statement um einen `case`-Label erweitern. So etwas kann man aber auch anders programmieren – und zwar so, dass Sie das Programm um neue Funktionen erweitern können, ohne diese Änderung durchzuführen. Diese Methode erlernen Sie gleich.

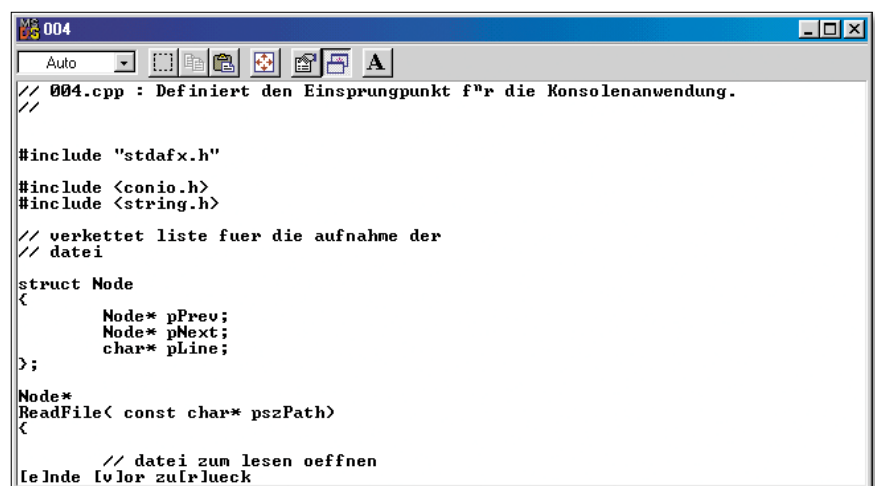
Benötigt wird also eine Organisationsmethode, bei der Sie einfach irgendwo eine Funktion implementieren können, die dann vom Rest des Programms automatisch benutzt wird. Das ist deutlich einfacher, als man sich das zunächst denken würde – allerdings ist die Art und Weise, wie man derlei Dinge in C/C++ hinschreiben muss, ein wenig gewöhnungsbedürftig.

Am einfachsten ist es, wenn man das Problem von der Seite angeht, die offensichtlich ist. In diesem Fall ist es die Art und Weise, wie eine Funktion und eine Taste miteinander verknüpft sind: Eine bestimmte Taste gehört immer zu einer bestimmten Funktion. Was liegt da näher, als das bereits bekannte "struct" zu verwenden, um einen entsprechenden Datentyp zu definieren, der genau diese Zuordnung widerspiegelt:

```
struct MenuEntry  
{  
    char cKey;  
    FunktionDieDazuGehört func;  
};
```

Ein Menü-Eintrag setzt sich also aus einer Taste bzw. einem Tastencode und einer Funktion zusammen. Aber was

dieses Arrays entspricht. Genauso ist das auch bei Funktionen: Der Name einer Funktion kann als Zeiger auf diese Funktion verwendet werden. Über diesen Zeiger können Sie die Funktion aufrufen. Bevor das aber passiert, müssen Sie noch etwas über Zeiger lernen – und zwar, wie man Zeiger dereferenziert.



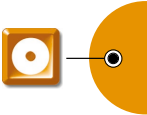
**DER DATEI-VIEWER** nimmt Formen an: Nun verfügt das Programm auch über eine Menüleiste.

Bisher haben Sie Zeiger zum Ablegen von Adressen von Nodes und zum Ablegen von Strings verwendet: Sie haben diese Zeiger aber nicht direkt benutzt, sondern diese stattdessen direkt an Funktionen aus der C-Laufzeitbiblio-

then den Inhalt der Speicherzelle, auf den der Zeiger zeigt – also das "H".

Dazu noch eine kleine Übung: Angenommen Sie haben folgenden Code:

```
char* pString = "Hallo Welt";  
char c1 = *pString;
```



```
putch( c1);
pString++;
char c2 = *pString;
putch( c2);
char c3 = *(pString + 3);
putch( c3);
```

Was wird hier am Bildschirm angezeigt? – Genau: "Hao". Das sollte aber vielleicht nochmals erläutert werden. Die Ausgabe des "H" ist vermutlich klar: Der Zeiger auf "Hallo Welt" wird dereferenziert, und darum erhält man das erste Zeichen, also das "H". Dann wird der Zeiger "pString" um eins inkrementiert, das bedeutet, pString zeigt nun auf

tionen haben mehrere Eigenschaften, die bei der Definition eines Zeigers alle berücksichtigt werden müssen: Im Wesentlichen sind das die Parameter der Funktion und der Rückgabewert der Funktion. Sie verwenden also nicht direkt einen "Zeiger auf eine Funktion", sondern einen "Zeiger auf eine Funktion, die einen bestimmten Parameter erhält und einen bestimmten Typ zurückliefert".

Nachdem das immer ein bisschen schwierig hinzuschreiben ist, verwendet man dazu in der Regel das "typedef"-

Diese Funktion erwartet keine Parameter() und liefert auch keinen Returnwert (void). Nochmals die Übersicht:

pFunc – der Name des Datentyps  
\* – der vom Typ Zeiger auf Funktion ist  
() – keine Parameter erwartet und  
void – nichts zurückliefert.

Diesen Datentyp könnten Sie nun benutzen. Um beispielsweise einen Zeiger auf die weiter oben definierte Funktion zu speichern und dann diese Funktion per Zeiger aufzurufen, müssten Sie folgenden Quellcode verwenden:

```
pFunc MeinZeiger = EineFunktion;
*pFunc();
```

Zunächst legen Sie also eine Variable vom Typ "pFunc" an und merken sich darin die Adresse der Funktion "EineFunktion", die Sie einfach anhand des Namens der Funktion ermitteln. Dann rufen Sie die Funktion auf, indem Sie den Zeiger dereferenzieren.

## Zur Praxis: das Menü

Die Funktionen zum Blättern im Dateibetrachter müssen ganz sicher mindestens einen Parameter haben, und zwar den Zeiger auf die "Erste anzuzeigende Zeile", denn der Parameter soll ja verändert werden. (Eigentlich muss etwas anderes übergeben werden, doch dazu später mehr). Die Funktionen sehen also in Anlehnung an den bereits bekannten Code etwa folgendermaßen aus:

```
void Forward( Node* pFirst)
{
    if( pFirst->pNext)
        pFirst = pNext;
}

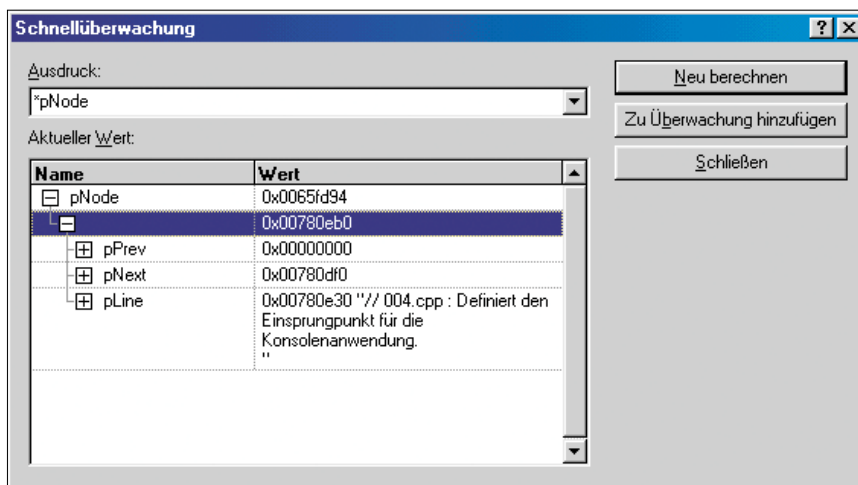
void Back( Node* pFirst)
{
    if( pFirst->pPrev)
        pFirst = pPrev;
}
```

Beide Funktionen haben also die gleiche Anzahl Parameter und die sind auch vom gleichen Typ. Außerdem liefern beide Funktionen das Gleiche zurück – und zwar nichts. Das macht die Funktionen zu perfekten Kandidaten für einen Funktions-Pointer-Typ, der per typedef wie folgt definiert werden kann:

```
typedef void(*pFunc)(Node*);
```

Sie haben also:

- einen Datentyp namens "pFunc"
- der vom Typ "Zeiger auf Funktion" ist (\*)
- wobei die Funktion einen Node\* als Parameter erhält und
- nichts (void) liefert.



**MIT DER SCHNELLÜBERWACHUNG** können Sie "auf die Schnelle" den Wert von Variablen überprüfen.

den Text "allo Welt". Wird dieser dereferenziert, so erhält man das "a". Dann wird der Zeiger nochmals um 3 erhöht (dabei ist zu beachten, dass er bereits auf das "a" und nicht auf das "H" zeigt) und das Resultat davon wird dereferenziert. Das Resultat davon ist das "o".

Allerdings kann und muss man die Sache mit den Zeigern manchmal noch ein bisschen weiter treiben. So gibt es auch Zeiger auf Zeiger und weitere mehrstufige Verwendungsmöglichkeiten. Diese werden Sie später im Beitrag auch noch benutzen.

Doch zurück zum Menü. Hier ging es darum, dass ein Menü-Eintrag einen Zeiger auf eine Funktion erhalten soll, und wie Sie diesen Zeiger zum Aufrufen der Funktion verwenden. Das Problem dabei ist zum einen das Dereferenzieren, zum anderen aber auch die Syntax, in der Sie einen Datentyp "Zeiger auf Funktion" definieren. Bei Zeigern auf andere Datentypen können Sie einfach einen Stern verwenden – char\* ist ein Zeiger auf Char. Bei Funktionen ist das etwas komplizierter, denn Funk-

tion. Damit definieren Sie einen neuen Datentyp, den Sie dann später unter dem selbst gewählten Namen verwenden können. Für einen Zeiger auf eine Funktion liegt zum Beispiel der Name "pFunc" (für Pointer to FUNCTION) nahe.

Angenommen, Sie haben folgende Funktion, auf die Sie einen Zeiger definieren wollen:

```
void EineFunktion()
{
    printf("In der Funktion");
}
```

Um für eine solche Funktion – also eine ohne Rückgabewert (void) und ohne Parameter einen Typ zu definieren, würden Sie den folgenden Ausdruck verwenden:

```
typedef void(*pFunc)();
```

Dieser Ausdruck ist ein bisschen kompliziert zu lesen, daher hier eine kurze Erläuterung. Zunächst einmal sehen Sie das "typedef"-Schlüsselwort: Es wird also ein Typ definiert. Dieser Typ soll den Namen "pFunc" haben und einen Zeiger auf eine Funktion (\*) darstellen.



Damit kann nun die "struct" für einen Menü-Eintragendgültig definiert werden:

```
struct MenuEntry
{
    char c;
    pFunc pF;
}
```

Nun setzt sich ein Menü einfach aus mehreren Menü-Einträgen zusammen: Was liegt da näher, als das Menü einfach als Array aus Menü-Einträgen zu definieren:

```
MenuEntry arrMenu[] =
{
    "v", Forward,
    "r", Back,
};
```

Sie definieren also ein Array aus "MenuEntry"s, und initialisieren dies mit zwei davon: Ein Eintrag ist fürs Vorwärtsblättern zuständig, der andere fürs Zurückblättern.

Was Sie nun noch benötigen, ist eine Funktion, die die richtige Funktion in Abhängigkeit der gedruckten Tasten ermittelt und aufruft (siehe Listing 1).

Das sieht komplizierter aus, als es tatsächlich ist. Die Funktion bekommt zwei Parameter: die gedruckte Taste und den Zeiger auf die erste anzuzeigende Zeile, der verändert werden soll. Dann iterieren Sie über alle Einträge in Ihrem Menü. Dabei machen Sie sich einen Trick zu Nutze: Der Compiler kann auch rechnen, und unter anderem ist eine der zur Verfügung stehenden Rechenoperationen das Berechnen der Größe von Datentypen und konstanten Daten. Das geht mit dem `sizeof()`-Operator zur Kompilierzeit.

So liefert `sizeof(arrMenu)` die Größe des Arrays "arrMenu" in Byte. Ebenso liefert `sizeof(MenuEntry)` die Größe eines Eintrages im Menü in Byte. Teilt man nun die Größe des Menüs durch die Größe eines Eintrags im Menü, erhält man die Anzahl an Einträgen im Menü – und das ist der Grenzwert bis zu dem iteriert werden soll. Der Trick dabei ist der, dass Sie das Menü auf diese Weise

einfach erweitern können, ohne den entscheidenden Quellcode zu verändern: Alles was Sie beim Hinzufügen einer neuen Funktion tun müssen, ist das "arrMenu" um einen Tastencode und einen Funktionsnamen zu erweitern, der Rest klappt dann automatisch.

In jedem Iterationsschritt vergleichen Sie nun die gedruckte Taste (c) mit der

len. Haben Sie aber keine Zeiger auf eine Struktur, sondern die Struktur selbst, verwenden Sie zum Erreichen der Member den "." Operator.

### Der Haken an der Sache

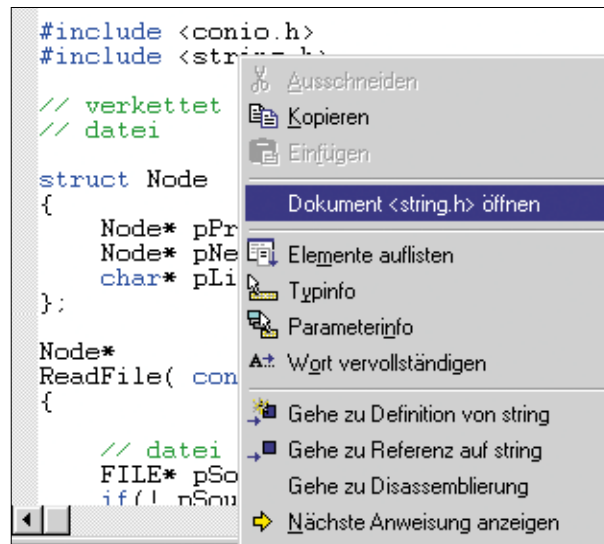
Es gibt aber einen klitzekleinen Haken am bisherigen Quellcode und das ist der folgende: Wenn Sie ihn ausprobieren, funktioniert er nicht.

Die Datei wird zwar weiterhin ganz brav angezeigt – nur das Scrollen funktioniert überhaupt nicht. Das sollten Sie zunächst einmal mit dem Debugger untersuchen. Setzen Sie dazu einen Breakpoint auf die Stelle in Ihrem Quellcode, wo Sie `FindAndExecCommand()` aufrufen. Starten Sie dann das Programm, und drücken Sie die Tasten zum Vorwärts-Scrollen. Der Debugger hält das Programm vor dem Aufruf von `FindAndExecCommand()` an.

Lassen Sie sich nun den Inhalt von `pFirst` anzeigen: Das ist die Zeiger-Variable, die auf die erste anzuzeigende Zeile zeigt und deren Wert beim Scrollen verändert werden soll. Die Adresse, auf die diese Variable zeigt, sollten Sie sich merken.

Dann arbeiten Sie die folgenden Quellcode-Zeilen interaktiv im Debugger ab. Das geht, indem Sie einfach Zeile für Zeile mit der Taste `[F11]` durchgehen: Der Debugger führt jeweils die aktuelle Zeile aus und wartet auf weitere Aktionen Ihrerseits.

Nach wenigen Schritten landen Sie dann in der Funktion `Forward()`. Auch hier lassen Sie sich den Wert von "pFirst" anzeigen – er entspricht logischerweise der Adresse, die Sie sich zuvor gemerkt haben. Dann wird in `Forward()` überprüft, ob es einen Nachfolger gibt – was der Fall ist. Deshalb wird `pFirst` auf diesen Nachfolger gesetzt. Erneut lassen Sie sich nun den Wert des Zeigers anzeigen – und siehe da: Er zeigt nun tatsächlich auf die Folgezeile. Nun gehen Sie die weiteren Programmschritte mit `[F11]` durch und landen in wenigen Schritten wieder im



**SIE KÖNNEN AUCH** die Inhalte der mitgelieferten Header-Dateien ansehen. Am einfachsten geht das über das Objekt-Menü im Quellcode-Editor.

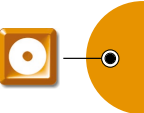
Taste aus dem gerade aktuellen Menü-Eintrag: `arrMenu[i].c`. Entspricht die gedruckte Taste der Taste, die dem Menü-Eintrag zugeordnet ist, rufen Sie einfach die dazugehörige Funktion auf, wobei Sie dieser noch den benötigten Parameter übergeben:

```
(*arrMenu[i].pF) (pFirst)
```

Hier sehen Sie zum ersten Male den Operator "." in Aktion. Dieser hat eine ähnliche Funktion, wie der Pfeil-Operator, den Sie schon kennen. Den Pfeil-Operator verwenden Sie, wenn Sie einen Zeiger auf eine Struktur haben und ein Member der Struktur erreichen wol-

### LISTING 1

```
void FindAndExecCommand( char c, Node* pFirst)
{
    for( int i=0; i<sizeof(arrMenu)/sizeof(MenuEntry); i++)
    {
        if( arrMenu[i].c == c)
        {
            (*arrMenu[ i].pF) ( pFirst);
        }
    }
}
```



Ausgangs Quellcode, unterhalb des Aufrufs von *FindAndExecCommand()*.

Erneut lassen Sie sich nun den Wert von "pFirst" anzeigen – und Überraschung: Die Variable hat wieder Ihren Ausgangswert, die Änderung die innerhalb von *Forward()* stattgefunden hat, scheint gar nicht eingetreten zu sein. Was ist hier geschehen?

## Call by Value vs. Call By Reference

Um das Problem zu verstehen, müssen Sie noch etwas über die Art und Weise der Parameterübergabe in C/C++ lernen.

Funktion unter dem Namen "x" angesprochen werden kann. Im Rahmen des Beispiels wird also *f2()* mit dem Wert 5 als Parameter aufgerufen. Innerhalb von *f2()* wird dann der Variablen *x* die initial den Wert 5 hatte, der Wert 6 zugewiesen. Dann wird *f2* wieder verlassen, und der Wert von "i" wird dem der Variablen "j" zugewiesen. Welchen Wert hat nun "j"? Ganz einfach: 5

Der Grund dafür liegt darin, dass *f2()* "per Wert" aufgerufen wurde. Das bedeutet, dass nicht die Variable "i" (also der Speicherbereich, in dem der Wert von *i* abgelegt wurde) an *f2* übergeben

```
int i = 5;
int* pi;
```

Wenn Sie nun möchten, dass die Zeiger-Variablen *pi* auf die Adresse von "i" zeigt, verwenden Sie den "&"-Operator wie folgt:

```
int* pi = &i;
```

Man nennt das "die Adresse nehmen" – "pi" nimmt also mit dem "&"-Operator die Adresse von "i". (Als kleiner Rückblick: Wenn Sie nun den Wert von "pi" ermitteln – was würden Sie dann erhalten? – Genau: die 5) Das Gleiche müssen Sie tun, wenn Sie möchten, dass eine Funktion den Wert einer Variablen ändern soll. Um beim *f1()*-, *f2()*-Beispiel zu bleiben, sähe die Sache wie folgt aus:

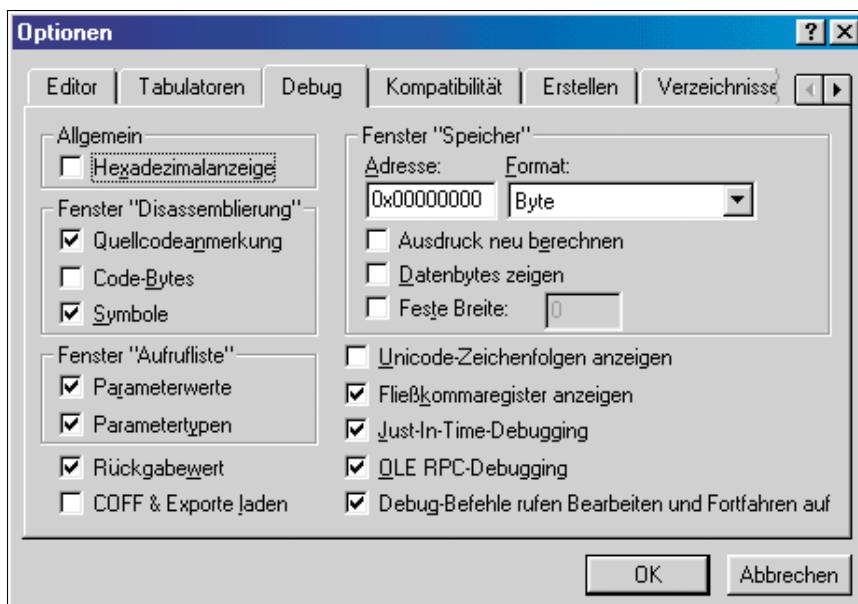
```
void f1()
{
    int i = 5;
    f2(&i);
    int j = i;
}
void f2(int* pi)
{
    *pi = 6;
}
```

Sie übergeben also die Adresse von "i" an *f2()*, und diese Funktion verwendet dann die Dereferenzierung, um in den Speicherbereich von "i" (der lokal als "pi" verwendet wird) den Wert "6" hineinzuschreiben. Genau das Gleiche muss auch im Datei-Viewer passieren. Allerdings kommt da ein kleines Problem dazu: Sie übergeben bereits einen Zeiger auf einen Node (Node\*). Sie möchten also nicht den Wert einer Variablen, sondern den Wert einer Zeiger-Variablen verändern.

Das geht aber genauso wie bereits gezeigt, nur müssen Sie dann entsprechend mehr \* einsetzen. Sie haben dann keinen Zeiger auf einen Typ mehr, sondern einen Zeiger auf einen Zeiger auf einen Typ: Node\*\* ppNode (Pointer to Pointer to Node).

Im fertigen Beispielprojekt auf der CD ist das bereits fertig ausprogrammiert: Nun wäre ein guter Zeitpunkt, dies einmal mit dem Debugger Schritt für Schritt nachzuvollziehen.

In diesem Beitrag haben Sie erfahren, wie Sie Zeiger auf Funktionen verwenden können – und dass Sie auch Zeiger auf Zeiger benutzen können. Außerdem haben Sie gesehen, dass die Verwendung von Zeigern zumindest dann unerlässlich ist, wenn Sie möchten, dass eine Funktion den Wert einer Variablen aus der aufrufenden Funktion verändern kann. Mit diesen Grundlagen gehen Sie nun ins nächste Kapitel. Dort festigen Sie Ihr Wissen, indem Sie mehr mit Zeigern herumoperieren und den Datei-Viewer mit zusätzlichen Funktionen ausstatten. UR



UNTER DEN EINSTELLUNGEN DER IDE sind auch eine ganze Reihe von Optionen für den Debugger zu finden. Dazu zählt zum Beispiel auch, in welchem Format die Speicherauszüge angezeigt werden sollen.

Es gibt zwei Arten, Parameter zu übergeben: "By Value" (per Wert) und "by Reference" (per Referenz). Im einen Fall ändert sich der Wert der Ausgangsvariable, und im anderen Fall nicht. Um das zu verdeutlichen, hier ein kleines Beispiel:

```
void f1()
{
    int i = 5;
    f2(i);
    int j = i;
}

void f2(int x)
{
    x = 6;
}
```

Sie haben also zwei Funktionen: *f1()* und *f2()*. In der Funktion *f1* wird zunächst eine Variable namens "i" mit dem Wert 5 belegt. Dann wird die Funktion *f2()* aufgerufen. Diese erwartet einen Parameter vom Typ "int", der innerhalb der

wurde, sondern einfach nur eine Kopie dieses Wertes in einem anderen Speicherbereich. Dieser andere Speicherbereich wird zwar in *f2()* mit dem Wert 6 überschrieben – nachdem es sich aber um einen anderen Bereich handelt als der für "i" reservierte, ändert das am Wert von "i" nichts.

Wenn Sie möchten, dass eine Funktion den Inhalt einer Variablen ändern kann, müssen Sie der Funktion die Adresse des Wertes übergeben – ansonsten kann die Funktion nicht den gewünschten Speicherbereich verwenden und damit auch den Inhalt der Ausgangsvariable nicht überschreiben. Die Adresse einer Variablen erhalten Sie mit dem "&"-Operator. Angenommen, Sie haben eine Integer Variable und einen Zeiger-Variable die auf "int" zeigt: