

C für Java Programmierer

Markus Thaler

März 2001, (Feb. 2002, Jul. 2004)

Die Programmiersprache C wurde Anfang der 70er zusammen mit dem Betriebssystem Unix entwickelt. Der ursprüngliche Gedanken war, portable, schnelle und kompakte Programme schreiben zu können. Das sind Eigenschaften, die man eigentlich nicht von einer einzigen Sprache erwarten darf. Schnelligkeit erwartet man am ehesten von maschinennaher Programmierung, Portabilität von höheren Programmiersprachen: C stellt aus dieser Sicht einen Kompromiss dar. Aber gerade wegen dieser Eigenschaften eignet sich C speziell für die Implementation von Betriebssystemen, wo vor allem Geschwindigkeit, aber auch Portabilität eine wesentliche Rolle spielen. Das Unix und Linux Betriebssystem z.B. ist zum grössten Teil (bis auf einige wenige Zeilen Assembler) in C geschrieben, Windows NT in C und C++. Systemnahe Programmierung erfolgt aus diesen Gründen auch meist in C/C++. C++ wurde Anfang 1980 mit dem Ziel, Objektorientierung einzuführen, entwickelt. Gleichzeitig musste aber Rückwärtskompatibilität zu C gewährleistet werden, d.h. jedes C-Programm kann auch mit einem C++ Compiler übersetzt werden.

Im Fach Betriebssysteme kommen wir deshalb nicht um C (C++) Programmierung herum, möchten aber aus verschiedensten Gründen (Komplexität, Unterschiede zu Java, etc.) nicht C++ einführen. Dies soll an entsprechender Stelle in der Vertiefung geschehen. Wir werden jedoch, soweit als möglich den C++ Compiler verwenden, da er vor allem bezüglich Syntaxüberprüfung Vorteile gegenüber einem reinen C Compiler aufweist.

Wir werden im folgenden vor allem auf die Unterschiede zwischen Java und C eingehen und die üblichsten "Trap Doors" bei C diskutieren. Viele Anmerkungen gelten auch für die C++ Programmierung. Weiter werden wir auf die Programmentwicklung von C-Programmen eingehen.

Die vorliegende Einführung ist in keiner Weise vollständig oder deckt sämtliche Aspekte der C-Programmierung ab, das ist auch nicht unser Ziel. Wir wollen Ihnen hier möglichst kompakt die allernotwendigsten Kenntnisse und Fertigkeiten vermitteln, um C-Programme verstehen und einfache Programme selber schreiben zu können.

Weitere Informationen zu C und C++ finden Sie in der Literatur:

B.W. Kernighan, D.M. Ritchie, Programmieren in C, Hanser Verlag, München.

B. Stroustrup, Die C++ Programmiersprache, Addison-Wesley, 1998.



If you cannot do in C, it can't be done !

Inhalt

Inhalt	2
1. Java und C: was ist (fast) gleich ?	3
1.2 Low-Level Syntax	3
1.3 Deklarationen	3
1.4 Zuweisungen in C	3
1.5 Input und Output	4
2. Java und C: was ist anders ?	6
2.1 Konstanten und Macros	6
2.2 Compiler Direktiven	7
2.2.1 Header Dateien	7
2.2.2 Bedingte Vorverarbeitung und Übersetzung	7
2.3 Strukturen und Typen	8
2.3.1 Struktur Typen	8
2.3.2 Zeichenketten resp. Strings	9
2.3.3 Aufzählungstypen	9
2.3.4 Vereinigungstypen	9
2.4 Zeiger	9
2.4.1 Zeiger auf primitive Datentypen	10
2.4.2 Parameterübergabe	10
2.4.3 Zeiger auf Strukturen	10
2.4.4 Zeiger und Arrays	11
2.4.5 Zeiger auf Funktionen (Funktionen als Argumente)	12
2.4.6 Argumente aus Kommandozeile	12
2.5 Memory Management	13
2.6 C-Programme	14
2.6.1 Aufbau eines Programmes in C	14
2.6.2 Modulare Programmierung	16
2.6.3 Übersetzen eines C (C++)-Programmes unter Unix / Linux	18
2.7 Make - Utility	19
2.7.1 Make, etwas fortgeschritten	20
2.8 Manual Pages und C-Funktionen	20

1. Java und C: was ist (fast) gleich ?

1.2 Low-Level Syntax

Anweisungen, Ausdrücke, Deklarationen und Kontrollstrukturen sind in C (C++) und Java sozusagen identisch und können in den meisten Fällen ohne Modifikation übernommen werden. Selbstverständlich fehlen in C sämtliche Sprachkonstrukte für die Definition von Klassen und Interfaces, zur Instanziierung von Objekten und zum Abfangen von Ausnahmen, etc.

1.3 Deklarationen

Verwenden Sie den C Compiler, müssen die Variablendeklarationen immer vor dem Programmcode stehen. Der C++ Compiler erlaubt hingegen die Deklaration der Variablen wo immer Sie wollen (wie in Java). Beispielsweise können Sie eine Laufvariable wie folgt deklarieren:

```
for (int j = 0; j < N ; j++) {
    . . .
}
```

In C gibt es keinen Typ `boolean`, dazu wird ein numerischer Datentyp (auch `float`) verwendet, wobei 0 als `false` und `!=0` als `true` definiert ist. D.h. aber auch, dass die Bedingungen bei `if`, `while`, etc. einen numerischen Wert enthalten können (das ist natürlich sehr schlechter Programmierstil und muss vermieden werden). Verwenden Sie also auf keinen Fall:

```
if (someInt)           sondern           if (someInt != 0)
```

Ein 8-Bit Zahl (ein Byte) wird wir in C als `char` deklariert, der im Normalfall ein ASCII Zeichen darstellt, jederzeit aber auch als 8-Bit Integer behandelt werden kann. D.h. Sie können ohne weiteres mit Werten vom Typ `char` rechnen.

In C kann jeder numerische Datentyp als `unsigned` deklariert werden. Beispielsweise liegt `int` (normalerweise) im Bereich -2^{31} bis 2^{31} , `unsigned int` jedoch im Bereich 0 bis 2^{32} .

Kommentare: in C steht Kommentar zwischen `/* ..` und `.. */`
bei C++ darf auch, wie in Java, der Doppelschrägstrich `//` verwendet werden

1.4 Zuweisungen in C

Der Zuweisungsoperator in C ist gleich wie in Java: das Gleichheitszeichen `"="`. In C kann eine Zuweisung aber auch als Ausdruck verwendet werden, der das "Resultat" weitergibt. Auf diese Weise können mehrere Variablen mit dem gleichen Wert initialisiert werden:

```
a = b = c = 13 ;      ist identisch mit      a = (b =(c = 13));
```

Zuerst wird die 13 der Variablen `c` zugewiesen. Der Zuweisungs-Ausdruck wiederum gibt den zugewiesenen Wert (die 13) weiter an `b` und analog an `a`.

Jetzt kommen wir zu einem der häufigsten Syntax Fehler in C. Betrachten Sie dazu folgenden Programmausschnitt:

```
int  a = 0, b = 1;
if (a = b) {
    body;
}
```

Nehmen Sie an der Programmierer wollte "a == b" schreiben, um auf Gleichheit zu testen, d.h. "body" soll nur ausgeführt werden, wenn a gleich b ist. Was geschieht aber in unserem Fall. Zuerst wird der Variablen a der Wert von b, nämlich 1 zugewiesen. Weil der Ausdruck "(a = b)" ungleich 0 (d.h. true) ist, wird "body" ausgeführt, zudem ist a nun gleich b . . . war das Absicht des Programmierers ?

In C muss die Grösse eines Array bei der Deklaration angegeben werden:

```
int  intArray[100];
```

Achtung, C (C++) überprüft zur Laufzeit keine Arraygrenzen, wie das Java macht, Sie können jederzeit schreiben `intArray[101] = 16;` . . . was das zur Folge hat, können Sie sich denken.

1.5 Input und Output

In C wird die Funktion `printf()` für die Ausgabe auf den Bildschirm verwendet. `printf()` akzeptiert eine variable Anzahl von Argumenten (Anzahl ≥ 1), dabei muss das erste Argument immer ein String sein. Die restlichen Argumente sind Werte für Kontrollstrukturen, die im String eingebettet sind. Für uns sind folgende Kontrollstrukturen von Interesse:

<code>%c</code>	ein Zeichen
<code>%s</code>	ein String
<code>%d</code>	ein Integer, als Dezimalzahl dargestellt
<code>%f</code>	eine Fließkommazahl
<code>%x</code>	ein Integer als Hexadezimalzahl dargestellt

Beispiel:

```
char  monat[4];
int   Jahr;
strcpy(monat, "Mar"); // C-Funktion zum Kopieren von Strings
Jahr = 2000;
printf("Jahr: %d, Monat: %s.\n", Jahr, monat);
```

Ausgabe:

```
Jahr: 2000, Monat: Mar.
```

Mit der Funktion `fprintf()` kann Text in eine Datei geschrieben werden. `fprintf()` benötigt als ersten Parameter einen Dateizeiger (Zeiger: siehe Abschnitt 2.4), der mit der Funktion `fopen()` erzeugt wird. Die restlichen Parameter bei `fprintf()` sind gleich wie bei `printf()`.

```
FILE *fz; // Dateizeiger
char *name = "Markus"; // Zeiger auf String "Markus"
fz = fopen("JustAFile", "r+"); // File öffnen, read und write
fprintf(fz, "Hallo %s\n", name); // in File schreiben
```

Die Funktion `scanf()` ist die zu `printf()` analoge Eingabe-Funktion. Sie stellt praktisch die gleichen Umwandlungen zur Verfügung, allerdings in umgekehrter Richtung.

Beispiel:

```
int i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name); // Daten lesen

Eingabe:      "25   54.3e-1  Dreifuss"
```

Die Eingabe weist der Variablen `i` den Wert 25 zu, `x` den Wert 5.43 und `name` die Zeichenkette `Dreifuss` mit abschliessendem `\0` Zeichen. Zwischen den Eingabefeldern können beliebig viele Leerstellen, Tabulatoren, Trennzeichen und Zeilenumschaltungen stehen.

Beachten Sie, dass die Variablen `i` und `x` mit einem `&` versehen sind, `name` jedoch nicht. `&i` heisst, dass wir der Funktion `scanf()` einen Zeiger resp. eine Referenz auf die Variable `i` übergeben (d.h. die "Adresse" von `i`). Bei `name` ist das nicht notwendig, weil Arrayvariablen per Definition Zeiger sind. Mehr dazu im Abschnitt über Zeiger.

Achtung:

- Wenn sie anstelle von 25 die Fließkommazahl 23.4 eingeben, wird der Punkt als Trennzeichen interpretiert und 23 an `i`, 4 an `x` zugewiesen, etc.
- Falls der String `name` zu kurz gewählt wird, können Daten überschrieben werden, da C (C++) Arraygrenzen zur Laufzeit nicht überprüft.

Zum Lesen aus Dateien steht die Funktion `fscanf()` zur Verfügung, es gelten die gleichen Bedingungen wie bei `fprintf()`.

Für Datei Input/Output mit rohen Daten stehen die Systemfunktionen `open()`, `read()` und `write()` zur Verfügung. Im Gegensatz zu `fopen()`, `fprintf()` und `fscanf()` arbeiten diese Funktionen mit einem Dateideskriptor vom Typ `int` und die Steuerparameter sind leicht verschieden. Für Details möchten wir auf die entsprechende Literatur und Manual Pages verweisen.

Weitere wichtige Eingabefunktion sind `getchar()` und `putchar()`. Die Funktion `getchar()` liest ein einzelnes Zeichen von der Eingabe und weist es der entsprechenden Variablen zu. Analog gibt `putchar()` ein Zeichen aus.

Beispiel:

```
#include <stdio.h>

int main(void) {
    char MyChar;
    while ((MyChar = getchar()) != EOF)
        putchar(MyChar);
}
```

In diesem Beispiel werden Zeichen von der Tastatur eingelesen und auf den Bildschirm ausgegeben, solange bis ein EOF gelesen wird (Ctrl-D). Beachten Sie wie die Zuweisung in der Bedingung der `while` Anweisung gleichzeitig als Ausdruck verwendet wird.

Beim Beispiel ist Ihnen sicher die erste Zeile aufgefallen: `#include <stdio.h>`. Mit dieser Anweisung werden die Bibliotheksfunktionen für die Ein- und Ausgabefunktionen eingebunden. Mehr zu `#include` finden Sie im Abschnitt "Compiler Direktiven".

In C++ stehen "streams" für die Ein- / Ausgabe zur Verfügung. Dazu müssen die Definitionen der Streamfunktionen eingebunden werden: `#include <iostream >`.

Beispiel für Ausgabe:

```
printf("Jahr %d, Monat %s.\n", Jahr, monat);
```

mit Streams:

```
cout << "Jahr " << Jahr << ", Monat " << monat << endl;
```

Beispiel für Eingabe:

```
scanf("%d %f %s", &i, &x, name);
```

mit Streams:

```
cin >> i >> x >> name;
```

Bitte beachten Sie, dass bei Streams kein Zeigeroperator (&) verwendet werden muss. Wir möchten Sie darauf aufmerksam machen, dass "streams" nur im Zusammenhang mit einem C++ Compiler verwendet werden können.

Achtung:

Auch hier muss name genügend lang sein.

2. Java und C: was ist anders ?

2.1 Konstanten und Macros

Konstanten werden mit `#define` deklariert:

```
#define MAX_SIZE 10
```

Die Anweisung `#define` ist eigentlich ein Makromechanismus. Überall wo der Name `MAX_SIZE` innerhalb seines Definitionsbereiches auftritt, wird er durch seine Definition (hier 10) ersetzt. Die Definition kann aber auch ein komplexer Ausdruck sein, z.B.

```
#define Square(a) a*a
```

Wenn Sie im Programm schreiben:

```
int j, resultat;  
.  
.  
.  
resultat = Square(j);
```

wird der Aufruf übersetzt in:

```
resultat = j*j;
```

Die "Übersetzung" wird vom C / C++ Preprozessor ausgeführt und erzeugt deshalb inline-Code.

2.2 Compiler Direktiven

Neben der `#define` Anweisung kennt der Preprozessor noch weitere Befehle. Für uns interessant sind das Einbinden von Header Dateien mit `#include` und die bedingte Übersetzung resp. Einbindung mit `#ifdef` und `#endif`.

2.2.1 Header Dateien

Bei der C-Programmierung werden externe Funktionen resp. ihre Definition, globale Definitionen und Systemfunktionen über sogenannte Header Dateien eingebunden. Wollen Sie z.B. die C++ Streambibliothek verwenden, muss in ihrer C / C++ Datei ganz am Anfang stehen:

```
#include <iostream >
```

Folgendes ist zu beachten:

- die Funktionen der Streambibliothek sind ab dieser Zeile bekannt, d.h. `#include` Anweisungen sollten immer ganz am Anfang einer Datei stehen
- der Definitionsbereich für Preprozessoranweisungen (`#include`, `#define`, etc) umfasst die aktuelle Source Code Datei
- die Klammern `<>` geben an, dass es sich um eine C (C++) Header-Datei handelt, die im Verzeichnis `/usr/include` gesucht wird. Schauen Sie sich dieses Verzeichnis und einige Dateien einmal an.
- Header Dateien enden normalerweise auf `".h"`
- eigene Header Dateien legen Sie mit Vorteil im gleichen Verzeichnis wie Ihren Code ab, eingebunden werden die Header Dateien mit:

```
#include "myDefinitions.h"
```

2.2.2 Bedingte Vorverarbeitung und Übersetzung

Bei der C- und speziell bei der C++ - Programmierung kann es vorkommen, dass eine Header Datei mehrfach importiert wird und deshalb Mehrfachdeklarationen auftreten, was der Compiler als Fehler meldet. Abhilfe kann hier durch "bedingte Vorverarbeitung" geschaffen werden. Ihre Header Dateien (hier z.B. für die Datei `my_include.h`) sollten deshalb immer nach folgenden Schema strukturiert werden:

```
#ifndef MY_INCLUDE_DEFS
#define MY_INCLUDE_DEFS

    Inhalt der Header Datei

#endif
```

Falls `MY_INCLUDE_DEFS` schon definiert ist, werden alle Zeilen zwischen `#ifndef` und `#endif` übersprungen. Falls `MY_INCLUDE_DEFS` noch nicht definiert ist, wird der Inhalt der Header Datei eingebunden und `MY_INCLUDE_DEFS` definiert.

Die Anweisungen `#ifdef` und `#endif` werden auch für die bedingte Kompilation verwendet, zum Beispiel um Code auf verschiedene Rechnerplattformen portieren zu können.

Beachten Sie, dass:

- Sie für die Bedingungsvariablen (hier `MY_INCLUDE_DEFS`) global eindeutige Namen verwenden müssen
- obige Zeilen am Anfang resp. am Ende Ihres Header Dateien stehen müssen

2.3 Strukturen und Typen

In C stehen keine Klassen mit Klassenvariablen zur Verfügung. Zusammengehörende Daten werden in C in einer Struktur zusammengefasst. Zusätzlich können Sie Strukturen auch als Typen deklarieren.

2.3.1 Struktur Typen

Zusammengehörende Daten werden in C in einer Struktur zusammengefasst. Ein Datum können Sie z.B. wie folgt deklarieren:

```
struct Datum {
    char *Monat;    // Zeiger auf einen String
    int  Tag;      // Tag als integer
    int  Jahr;     // Jahr als Integer
};
```

Die Datenstruktur wird dann wie folgt instanziiert:

```
struct Datum MeinGeburtstag;
```

Die einzelnen Felder werden wie folgt gesetzt, resp. gelesen:

```
MeinGeburtstag.Monat = MonatMaerz; // Zuweisung eines Zeigers
MeinGeburtstag.Tag = 13;
MeinGeburtstag.Jahr = 1955;
```

`MonatMaerz` ist hier ein Zeiger auf einen String, der "Maerz" enthält, und kann z.B. wie folgt definiert werden (über den * unterhalten wir uns später: Abschnitt 2.4):

```
char *MonatMaerz = "Maerz"; // Zeiger auf den String "Maerz"
                          // fuer den String "Maerz" wird
                          // Speicher reserviert (alloziert)
```

Die Struktur `Datum` kann auch als Typ definiert werden:

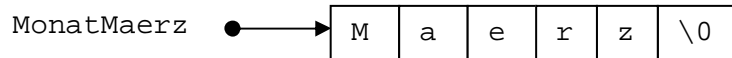
```
typedef struct{
    char *Monat;           // Zeiger auf einen String
    int  Tag;
    int  Jahr;
} Datum;
```

`Datum` kann nun wie ein üblicher (vordefinierter) Datentyp verwendet werden:

```
Datum MeinGeburtstag;
Datum ArrayVonDatuemmern[100]; // Array von Strukturen
```

2.3.2 Zeichenketten resp. Strings

In C (C++) werden Zeichenketten resp. Strings immer mit einem Null-Zeichen (ASCII-Wert 0) abgeschlossen. Der String "Maerz" aus obigem Beispiel hat folgende Form:



2.3.3 Aufzählungstypen

Im Gegensatz zu Java gibt es die sehr praktischen Aufzählungstypen. Mit Aufzählungstypen können Sie einen eigenen Datentyp mit einer diskreten Menge von Werten deklarieren:

```
enum PrimaerFarbe { blau, rot, gelb };

PrimaerFarbe MyColor;
. . .
MyColor = blau;
```

2.3.4 Vereinigungstypen

Vereinigungstypen erlauben das Speichern verschiedenster Datentypen in einer Speicherzelle. Ein Vereinigungstyp wird ähnlich wie eine Struktur deklariert:

```
union StringIntFloat {
    char *someString;    // Zeiger auf char
    int someInt;        // ein Integer
    float someFloat;    // ein Float
} SIFUnion;

SIFUnion EinBehaelter;
. . .
EinBehaelter.someInt = 330013;
```

Die Vereinigung führt nicht Buch, welcher Datentyp gerade gespeichert ist. Vereinigungstypen sollten wenn möglich vermieden werden. Sie sind fehleranfällig, weil beliebige Datentypen gemischt werden können und sind hier nur der Vollständigkeit halber aufgeführt (Sie werden sie aber im Zusammenhang mit Systemfunktion antreffen).

2.4 Zeiger

In Java sind Referenzen auf Objekte eigentlich Zeiger auf diese Objekte (in Assemblerjargon: "Adresse des Objekts"). Referenzen auf Variablen mit primitiven Datentypen wie `int`, `char`, etc. liefern immer den Wert der Variablen:

```
int    j, AnyVar;
j      = 13;
AnyVar = j;
```

Der Variablen `AnyVar` wird der Wert (Inhalt) von `j` zugewiesen, nämlich 13.

2.4.1 Zeiger auf primitive Datentypen

Variablen in C (C++) enthalten Datenwerte des definierten Typs. In C (C++) können auch explizit Zeiger auf Datenwerte (Typ: Zeiger auf Typ) definiert werden:

```
int x, *y; // x: Variable, *y: Zeiger auf int Variable
```

x ist eine Variable, die einen int-Wert enthält, y ist eine Variable, die einen Zeiger auf einen int-Wert enthält. Den Zeiger auf eine Variable erhält man mit dem Operator &:

```
y = &x; // &x: Adresse von x (Zeiger auf Variable x)
```

y enthält nun einen Zeiger, der auf den Datenwert x zeigt. Wenn dieser Zeiger bekannt ist, können Sie über den Operator * auf den Datenwert zugreifen (dereferenzieren des Zeiger) :

```
int someInt;
someInt = *y; // *y: Wert der Variablen auf die y zeigt
```

Ein "leerer" Zeiger, d.h. ein Zeiger der nirgendwohin zeigt, ist (wird) normalerweise auf 0 gesetzt. Ein solcher Zeiger heisst Nullzeiger (Null-Zeiger) und wird oft mit der Konstanten NULL beschrieben.

2.4.2 Parameterübergabe

In C werden Funktionsparameter generell als Werte übergeben. Variablen, deren Werte durch die Funktion verändert werden sollen, müssen über Zeiger übergeben werden.

Beispiel: die Funktion swap(a, b) vertauscht den Inhalt der beiden Variablen a und b

```
void swap(int *a, int *b) {
    int tmp;
    tmp = *a; // Inhalt von a zwischenspeichern
    *a = *b; // Inhalt von b nach a kopieren
    *b = tmp; // b mit a (in tmp) füllen
}
```

Aufruf der Funktion:

```
int val1, val2;
...
val1 = 15; val2 = 16;
swap(&val1, &val2);
printf("val1: %d, val2: %d\n", val1, val2);
```

Ausgabe:

```
val1: 16, val2: 15
```

2.4.3 Zeiger auf Strukturen

Betrachten wir als nächstes unser Datumsbeispiel. Wir definieren zusätzlich die Zeigervariable PtrAufDatum, die einen Zeiger auf eine Struktur vom Typ Datum enthält:

```

typedef struct{
    char *Monat;    // Zeiger auf String
    int  Tag;
    int  Jahr;
} Datum;

Datum *PtrAufDatum;
Datum  MeinGeburtstag;

PtrAufDatum = &MeinGeburtstag;

```

Der Zeiger wird mit Hilfe der "-> Syntax" dereferenziert, d.h. Sie können folgendermassen auf die einzelnen Felder der Struktur zugreifen:

```

PtrAufDatum->Monat = MonatMaerz;
PtrAufDatum->Tag = 13;
PtrAufDatum->Jahr = 55;

```

2.4.4 Zeiger und Arrays

Arrayvariablen sind in C (C++) grundsätzlich Zeiger auf den Array:

```

int  MyArray[13], *IntPtr, Val;
. . .
IntPtr = MyArray;
Val = *IntPtr; // Val wird der Wert von MyArray[0] zugewiesen

```

MyArray ist ein Zeiger auf den Datenwert an der Stelle MyArray[0]. MyArray kann einem Zeiger auf einen int-Wert (IntPtr) zugewiesen werden. IntPtr zeigt dann auf den ersten int-Wert des Arrays.

Arrays werden wie folgt an Funktionen übergeben

```

int  MyArray[13], Resultat;
. . .
Resultat = AddElemente(5,2, MyArray);

```

In der Funktionsdeklaration muss der Array als Zeiger auf den entsprechenden Elementtyp deklariert werden:

```

int AddElement(int a, int b, int *AnyArray){
    int retVal;
    retVal = AnyArray[a] + AnyArray[b];
    return(retVal);
}

```

Beachten Sie, dass in der Funktionsdeklaration keine Klammern [] vorkommen (die Arrayvariable ist ein Zeiger) und erinnern Sie sich, Grenzen werden keine überprüft.

Wenn Sie verhindern wollen, dass die Arraydaten in einer Funktion modifiziert werden, muss der Array als const deklariert werden:

```

int AddElement(int a, int b, const int *AnyArray) {

```

Vielleicht treffen Sie auf folgendes Konstrukt:

```
char **strArray;
```

das ist ein Zeiger auf einen Zeiger auf Zeichen oder auch ein variabel langer Array von Strings. Wir möchten hier nicht weiter auf Details eingehen, weil wir nun das unschöne Gebiet der Zeigerarithmetik betreten würden, ein Thema, das wir Ihnen im Moment lieber vorenthalten möchten.

2.4.5 Zeiger auf Funktionen (Funktionen als Argumente)

Manchmal ist es notwendig, dass eine Funktion als Parameter an eine andere Funktion übergeben wird, Sie werden dies im Praktikum zu Threads benutzen. Eine Funktion selbst ist keine Variable, sondern eigentlich ein Zeiger auf die Funktion. Dazu ein Demonstrationsbeispiel mit zwei alternativen Formulierungen (älter in Box):

```
void YesOrNo(int a, int b, int comp(int a, int b) ) {
    int yon;
    yon = comp(a,b);
    if (yon > 0)
        printf("Yes\n");
    else
        printf("No\n");
}

int Greater(int a, int b) {
    if (a > b) return(1);
    else      return(0);
}

int LowerEqual(int a, int b) {
    if (a <= b) return(1);
    else      return(0);
}
```

Im Hauptprogramm wird die Funktion YesOrNo mit den Funktionen Greater resp. LowerEqual als Parameter wie folgt aufgerufen:

```
int Val1 = 0, Val2 = 45, Val3 = 88;
. . .
YesOrNo(Val1, Val2, Greater);
YesOrNo(Val2, Val3, LowerEqual);
```

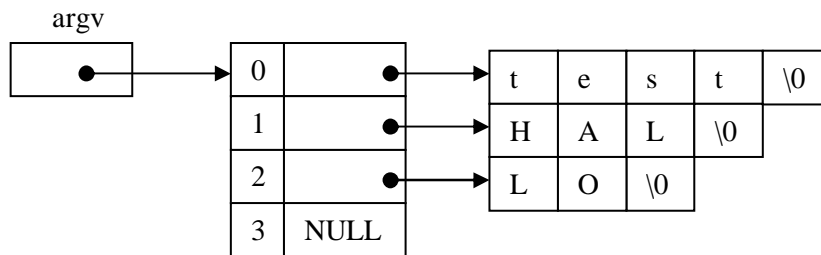
Ausgabe: No
 Yes

2.4.6 Argumente aus Kommandozeile

Programme werden oft mit Argumenten gestartet. Argumente werden über zwei Parameter zur Verfügung gestellt (hier sehen Sie eine Anwendung von **):

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

argc enthält die Anzahl Argumente, argv einen Zeiger auf einen Array von Strings:



Der letzte Eintrag im Array (argv[3]) ein NULL Zeiger und jeder String ist, wie in C üblich, mit einem \0 Zeichen (Nullcharacter) abgeschlossen. Der Eintrag bei argv[0] zeigt immer auf den Namen des Programms. argc ist in diesem Beispiel 3.

Im Hauptprogramm können Sie z.B. folgendermassen auf die Argumente zugreifen:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int j;
    if (argc > 0) {
        printf("Programm: %s\n", argv[0]);
    }
    for (j = 1; j < argc; j++) {
        printf("%s", argv[j]);
    }
    printf("\n");
}
```

Rufen Sie das Programm wie folgt auf:

```
test HAL LO
```

sehen Sie auf dem Bildschirm:

```
Programm: test
HALLO
```

2.5 Memory Management

Java ist eine "garbage-collected" Sprache, d.h. Sie müssen sich im Allgemeinen nicht um die dynamische Allokierung und Freigabe von Speicher kümmern, ganz im Gegensatz zu C (C++). Wir beschränken uns hier auf die dynamische Speicherallokierung in C (hier gibt es grosse Unterschiede zu C++).

Betrachten Sie folgende Anweisungen:

```
#include <stdlib.h>
#define NUM_CHARS 256
char *Monat;
. . .
Monat = (char *) malloc(NUM_CHARS * sizeof(char));
strcpy(Monat, "Monat Maerz");
```

Die Funktionen für dynamische Speicherallozierung sind in `stdlib.h` definiert und werden über die Anweisung `#include` eingebunden. In der zweiten Zeile wird eine Konstante definiert, die die maximale Anzahl Zeichen pro String festlegt, dann wird ein Zeiger auf ein Zeichen (resp. Zeichenkette) deklariert. In der fünften Zeile wird dynamisch Speicher mit der Funktion `malloc()` angefordert (hier 256 mal die Grösse eines `char`). Die Funktion `malloc()` gibt als Resultat eine Zeiger auf den allozierten Speicherbereich zurück und muss mit einem Cast auf den richtigen Typ gesetzt werden. Die Funktion `sizeof()` gibt die Länge eines Datentyps in Anzahl Bytes zurück. Nachdem Speicherplatz bereitsteht, kann mit der Systemfunktion `strcpy()` der konstante String "Monat Maerz" in den String `Monat` kopiert werden. Beachten Sie bitte, dass jeder String in C(C++) mit einer `\0` terminiert ist, also einen Speicherplatz vom Typ `char` mehr als die Anzahl sichtbarer Zeichen benötigt.

Weil C keine Garbage Collection kennt, müssen Sie nicht benötigten Speicherplatz auch selbst wieder freigeben, dies geschieht mit der Funktion `free()` mit dem Zeiger auf den Speicherplatz als Argument, in unserem Beispiel:

```
free(Monat);
```

Selbstverständlich können Sie auf diese Art und Weise Speicher für beliebige Datenstrukturen dynamisch allozieren und deallozieren.

2.6 C-Programme

Den Quellcode für die im folgenden verwendeten Programmbeispiele `ynMain.cc` und `ynMainModule.cc` finden Sie auf dem BS WEB-Server www-t.zhwin.ch/ki/bs unter Praktika im Archiv `gettingFamiliar.tar.gz`. Nach dem Auspacken des Archivs (siehe Anleitung auf dem WEB-Server) finden Sie die Dateien im Ordner `CforJ/ynSingle` resp. in `CforJ/ynModule`.

2.6.1 Aufbau eines Programmes in C

Auf der nächsten Seite finden Sie den möglichen Aufbau eines C-Programmes, abgelegt in einem einzigen File. Da wir meistens den C++ Compiler verwenden, müssen die Quellendateien entweder auf `.cc` oder `.cpp` enden (bei reinen C-Programmen genügt ein `.c`). Es gibt zwei Möglichkeiten, um Funktionen zu spezifizieren: entweder wird eine Funktion vor dem Hauptprogramm oder nach dem Hauptprogramm eingefügt. In letzterem Fall muss aber vor dem Hauptprogramm die Funktion mit einem sogenannten "Funktionsprototyp" bekannt gemacht werden:

```
int LowerEqual(int a, int b);
```

Hier noch die Header Datei `myDefinitions.h`:

```

/*****
*      File:   myDefinitions.h
*      Autor:  M. Thaler
*      Datum:  Jan. 2000 / Jul. 2004
*****/

#ifndef MY_DEFINITIONS
#define MY_DEFINITIONS

#define YES    8888 /* just a figure */
#define NO    9999 /* just a figure */

#endif

/*****/

```

```

/*****
*   File:   ynMain.cc
*   Autor:  M. Thaler
*   Datum:  Jan. 2000, Feb. 2001, Jul. 2004
*****/

#include <stdio.h>
#include "myDefinitions.h"

/*****

#define CONST_A      0
#define CONST_B      33
#define EIGHT_EIGHT 88

/*****

char  *YesStr = "Yes", *NoStr = "No";
int   aGlobalVar;

/*****

int YesOrNo(int a, int b, int comp(int a, int b) ) {
    int YON;
    YON = comp(a,b);
    if (YON > 0)
        return YES;
    else
        return NO;
}

int Greater(int a, int b) {
    if (a > b)
        return(1);

    else
        return(0);
}

int LowerEqual(int a, int b);

/*****

int main(void) {

    int a = CONST_A, b = CONST_B;

    aGlobalVar = EIGHT_EIGHT;
    if (YesOrNo(a, b, Greater) == YES)
        printf("%d > %d ?\t%s\n", a, b, YesStr);
    else
        printf("%d > %d ?\t%s\n", a, b, NoStr);
    if (YesOrNo(b, aGlobalVar, LowerEqual) == NO)
        printf("%d <= %d ?\t%s\n", b, aGlobalVar, NoStr);
    else
        printf("%d <= %d ?\t%s\n", b, aGlobalVar, YesStr);
}

/*****

int LowerEqual(int a, int b) {
    if (a < b)
        return(1);
    else
        return(0);
}

/*****

```

Header Text mit Informationen zum Programm

Einbinden der Header Datei

Definition von Konstanten (ev. auch in der Header Datei myDefinitions.h)

Hier definierte Variablen sind innerhalb dieses Programmes global bekannt

Implementation der Funktionen : YesOrNo(), Greater()

Funktionsprototyp der Funktion : LowerEqual(), notwendig, weil Implementation erst nach main().

Hauptprogramm, hier ohne Argumente

Implementation der Funktion LowerEqual()

2.6.2 Modulare Programmierung

Im allgemeinen sind die Programme nicht so kurz wie ynMain. Oft ist es sinnvoll, Programme in verschiedene Module aufzuteilen, die auch einzeln kompiliert werden können. Für unsere Anwendung würde sich z.B. folgende Aufteilung anbieten:

Modultyp	Filename	Funktionen	Variablen
Hauptmodul	ynMainModule.cc	main()	globale Variablen
Unterm modul 1	yesorno.cc yesorno.h	YesOrNo()	
Unterm modul 2	compare.cc compare.h	Greater() LowerEqual()	
Definitionen	myDefinitions.h		Konstante MAX_CHAR

Nun werden Sie sich fragen wieso wir die zwei zusätzlichen Header Files yesorno.h und compare.h eingeführt haben. In diesen Files machen wird dem Hauptprogramm die in den Modulen definierten Funktionen bekannt. Untenstehend finden Sie die Implementation der einzelnen Files.

Datei: ynMainModule.cc

```

/*****
*      File: ynMainModule.cc
*      Autor: M. Thaler
*      Datum: Jan. 2000 / Jul. 2004
*****/

#include <stdio.h>
#include "myDefinitions.h"
#include "yesorno.h"
#include "compare.h"

/*****

#define CONST_A          0
#define CONST_B          33
#define EIGHT_EIGHT 88

/*****

char  *YesStr = "Yes", *NoStr = "No";
int   aGlobalVar;

/*****

int main(void) {

    int a = CONST_A, b = CONST_B;

    aGlobalVar = EIGHT_EIGHT;
    if (YesOrNo(a, b, Greater) == YES)
        printf("%d > %d: %s\n", a, b, YesStr);
    else
        printf("%d > %d: %s\n", a, b, NoStr);
    if (YesOrNo(b, aGlobalVar, LowerEqual) == NO)
        printf("%d <= %d: %s\n", b, aGlobalVar, NoStr);
    else
        printf("%d <= %d: %s\n", b, aGlobalVar, YesStr);
}

/*****

```

Datei: yesorno.h

```
/*
 * File: yesorno.h
 * Autor: M. Thaler
 * Datum: Jan. 2000 / Jul. 2004
 */

#ifndef YESORNO_HEADER
#define YESORNO_HEADER

int YesOrNo(int a, int b, int (*comp)(int a, int b));

#endif

/*
```

Datei: yesorno.cc

```
/*
 * File: yesorno.cc
 * Autor: M. Thaler
 * Datum: Jan. 2000, Feb. 2001, Jul 2004
 */

#include <stdio.h>
#include "myDefinitions.h"

/*

int YesOrNo(int a, int b, int comp(int a, int b) ) {
    int yon;
    yon = comp(a,b);
    if (yon > 0)
        return YES;
    else
        return NO;
}

*/
```

Datei: compare.h

```
/*
 * File: compare.h
 * Autor: M. Thaler
 * Datum: Jan. 2000 / Jul 2004
 */

#ifndef COMPARE_HEADER
#define COMPARE_HEADER

int Greater(int a, int b);
int LowerEqual(int a, int b);

#endif

/*
```

Datei: compare.cc

```

/*****
*   File: compare.cc
*   Autor: M. Thaler
*   Datum: Jan. 2000, Jul. 2004
*****/

int Greater(int a, int b) {
    if (a > b)
        return(1);
    else
        return(0);
}

int LowerEqual(int a, int b) {
    if (a < b)
        return(1);
    else
        return(0);
}

/*****/

```

Datei: myDefinitions.h

```

/*****
*   File: myDefinitions.h
*   Autor: M. Thaler
*   Datum: Jan. 2000 / Jul. 2004
*****/

#ifndef MY_DEFINITIONS
#define MY_DEFINITIONS

#define YES 8888 /* just a figure */
#define NO 9999 /* just a figure */

#endif

/*****/

```

2.6.3 Übersetzen eines C (C++)-Programmes unter Unix / Linux

yorno.cc können Sie wie folgt mit dem C++ GNU Compiler übersetzen:

```
g++ yorno.cc -o yorno
```

Das Programm kann durch Eingeben von yorno resp. ./yorno gestartet werden.

Etwas aufwendiger ist die Übersetzung von yornom.cc. Hier müssen zuerst die einzelnen Module übersetzt werden, dabei werden als Resultat die Object-Dateien mit den Endungen .o erzeugt:

```
g++ -c yesorno.cc [-o yesorno.o]
g++ -c compare.cc [-o compare.o]
g++ -c ynMainModule.cc [-o ynMainModule.o]
```

Optionen: -c : Übersetzen ohne zu linken, -o : Name der Ausgabedatei, []: ist optional.

Die Object Dateien müssen dann endgültig zu einem lauffähigen Programm gelinkt werden, hier muss die Option `-c` weggelassen werden:

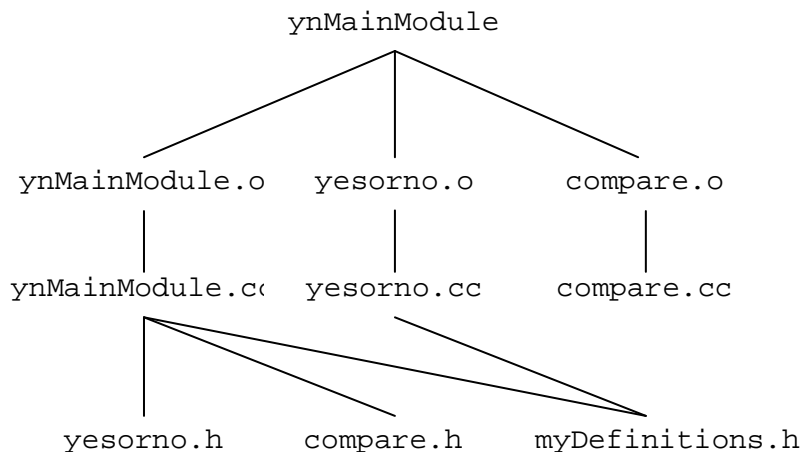
```
g++ yesorno.o compare.o ynMainModule.o -o ynMainModule.e
```

Das Programm kann durch Eingeben von `ynMainModule.e` resp. `./ynMainModule.e` gestartet werden.

Die Übersetzung einzelner Source Dateien auf diese Art und Weise ist natürlich sehr ineffizient, deshalb steht unter Unix / Linux die **make** Utility zur Verfügung.

2.7 Make - Utility

Für die modulare Programmversion von `yorno` können wir leicht die Abhängigkeiten zwischen den einzelnen Dateien aufzeichnen. Das ausführbare Programm `yornom` baut auf den Object Dateien `yornom.o`, `yon.o` und `cmp.o` auf, die Object Dateien selbst hängen von den Source Code Dateien `*.cc`, diese wiederum von den entsprechenden Header-Dateien:



Machen Sie z.B. eine Änderung am File `myDefinitions.h`, dann hat das sowohl Auswirkungen auf `yon.cc` als auch auf `yornom.cc`, d.h., Sie müssen `yon.cc` und `yornom.cc` neu übersetzen und anschliessend neu linken ... eine mühsame Arbeit.

Dieses Vorgehen lässt sich unter Unix / Linux mit Hilfe der Make Utility automatisieren. Dazu muss in einer Datei mit Namen `makefile` der Abhängigkeits-Baum in textueller Form formuliert werden:

```

ynMainModule:      ynMainModule.o yesorno.o compare.o myDefinitions.h
                      g++ ynMainModule.o yesorno.o compare.o -o ynMainModule.e

ynMainModule.o:   ynMainModule.cc myDefinitions.h
                      g++ -c ynMainModule.cc -o ynMainModule.o

Target — yesorno.o:  yesorno.cc myDefinitions.h
                      g++ -c yesorno.cc -o yesorno.o

compare.o:       compare.cc
                      g++ -c compare.cc -o compare.o — Befehlszeile
  
```

Die fetten Zeilen beschreiben die Abhängigkeiten entsprechend unserem Baum. Die restlichen Zeilen sind Befehle, die ausgeführt werden müssen, um das jeweilige **Target** (Knoten im Baum) neu zu bilden (resp. zu übersetzen)

Wenn Sie das Programm `make` starten, wird zuerst das `makefile` eingelesen, der Baum aufgestellt und anhand der Modifikationszeiten der Dateien festgestellt, ob eine Datei neuer als die Wurzel (hier `yornom`) des Baumes ist. Wenn ja, entscheidet `make` aufgrund des Baumes, welche Dateien neu übersetzt und gelinkt werden müssen. Dazu werden die entsprechenden Befehlszeilen ausgeführt.



Beachten Sie, dass für die Abstände zwischen Targets und, Abhängigkeiten resp. Befehlszeilen Tabulatoren verwendet werden müssen, sonst terminiert make mit einer Fehlermeldung.

2.7.1 Make, etwas fortgeschritten

Make bietet selbstverständlich mehr Möglichkeiten als oben beschrieben. Ein etwas erweitertes `makefile` für `yornom` sieht etwa so aus:

```

CMP=      g++
CMPFLAGS= -g -c

yornom:   yornom.o yon.o cmp.o
          $(CMP) yornom.o yon.o cmp.o -o yornom.e

yornom.o: yornom.cc my_defs.h
          $(CMP) $(CMPFLAGS) yornom.cc

yon.o:    yon.cc my_defs.h
          $(CMP) $(CMPFLAGS) yon.cc

cmp.o:    cmp.cc
          $(CMP) $(CMPFLAGS) cmp.cc

clean:
          rm *.o

all:
          touch *.cc
          make

```

Neu sind hier Macros dazugekommen, die in den Zeilen 1 und 2 definiert werden und in den jeweiligen Befehlszeilen angewendet werden. In Zeile 2 z.B. werden die Compileroptionen gesetzt. Das `-g` bedeutet, dass beim Übersetzen Zusatzinformation für den Debugger erzeugt werden soll. Ist das Programm soweit entwickelt, dass dies nicht mehr notwendig ist, können Sie einfach die Option `-g` löschen und das Programm neu übersetzen. Würden Sie jetzt aber `make` eingeben, erhalten Sie die Meldung "Program uptodate". Um dies zu verhindern müssen Sie `make all` eingeben. Make springt nun zum Target `all` und führt dort die geforderten Befehle aus: `touch *.cc` und dann `make`. `touch` ist eine Unix Systembefehl und setzt einfach die Modifikationszeit aller angegebenen Dateien auf die aktuelle Zeit. Das nachfolgende `make` wird deshalb das gesamte Programm neu übersetzen.

Moderne Benutzeroberflächen unter Linux, wie z.B. KDE, bieten für die Programmentwicklung integrierte Entwicklungsumgebungen mit ähnlichen Eigenschaften wie Kawa oder Visual Studio an. Im Zusammenhang mit Systemprogrammierung und Unix werden Sie jedoch noch öfters `makefiles` antreffen.

2.8 Manual Pages und C-Funktionen

Die Manual Pages der C-Funktionen können Sie mit dem Unix/Linux-Befehl `man` (`xman`) abrufen. Die C-Funktionen finden Sie in den Sections 2 und 3. Doppelklicken Sie den entsprechenden Eintrag zu Anzeigen der benötigten Manual Page. Ein guter Startpunkt für die Erkundung möglicher C-Funktionen sind auch die Header Dateien, die Sie unter Unix in `/usr/include` und den entsprechenden Unterverzeichnissen finden.