

[C/C++ Forum](#) :: [Die Artikel](#) :: [Sockets und das HTTP-Protokoll](#)

Gehen Sie zu Seite [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#) [Weiter](#) [Zeige alle Beiträge auf einer Seite](#)

joomla Autor 20:10:46 09.01.2007 Titel: **Sockets und das HTTP-Protokoll**

1 Vorwort

1.1 Einleitung

Willkommen zu meinem ersten Tutorial, in dem ich erklären möchte, wie man Sockets unter Linux und Windows benutzt. Am Ende werden wir ein Programm geschaffen haben, welches, auf Eingabe einer URL, über das HTTP-Protokoll die entsprechende Datei herunterlädt und auf der Festplatte speichert. Sie merken, es geht in diesem Tutorial nur um das Verbinden, also fangen Sie am Besten nicht an mit lesen, wenn Sie eine Einführung in die Programmierung von Netzwerkspielen erwarten.

1.2 Voraussetzungen

Unter Windows benutze ich als IDE Code::Blocks (<http://www.codeblocks.org>) und den GNU-GCC-Compiler in der Version 3.4.4. Unter Linux verwende ich gedit (<http://www.gnome.org/projects/gedit/>) und den GNU-GCC-Compiler in der Version 4.1.2.

Sie sollten C++-Kenntnisse in Exceptions, Stringstreams und Filestreams haben, außerdem ist Erfahrung mit Zeigern empfehlenswert.

1.3 Linken der benötigten Libs

Wenn Sie unter Linux sind, brauchen Sie gar nichts tun. Falls Sie aber unter Windows arbeiten, müssen Sie noch gegen die benötigte [Winsock](#)-Library linken. Der Name der Lib lautet libws2_32.a. Falls diese bei Ihnen nicht vorhanden ist, suchen Sie einfach eine beliebige heraus, die nach Sockets klingt und probieren Sie sie einfach aus. Sollte das nicht helfen, fragen Sie am besten im Compiler- oder WinAPI-Forum. Hier 2 Screenshots für das Hinzufügen mit **Code::Blocks**:

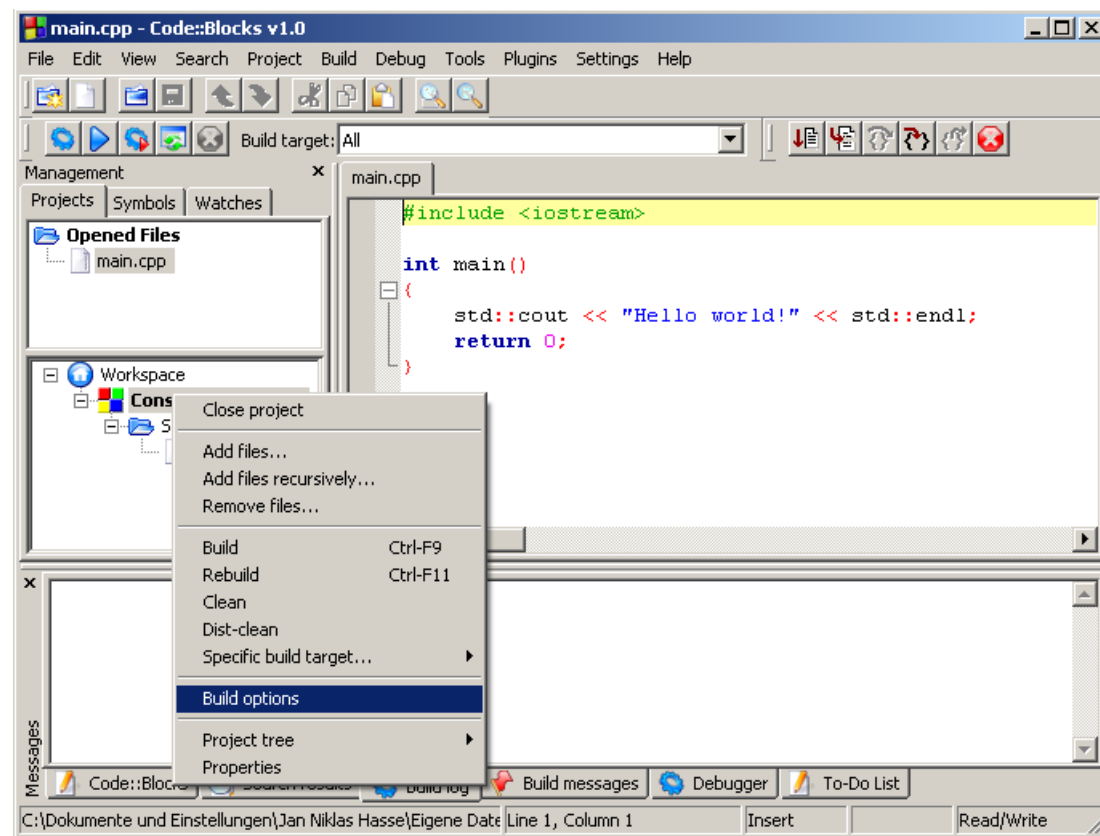


Abb. 1.3.1 Auswählen der Build-Options

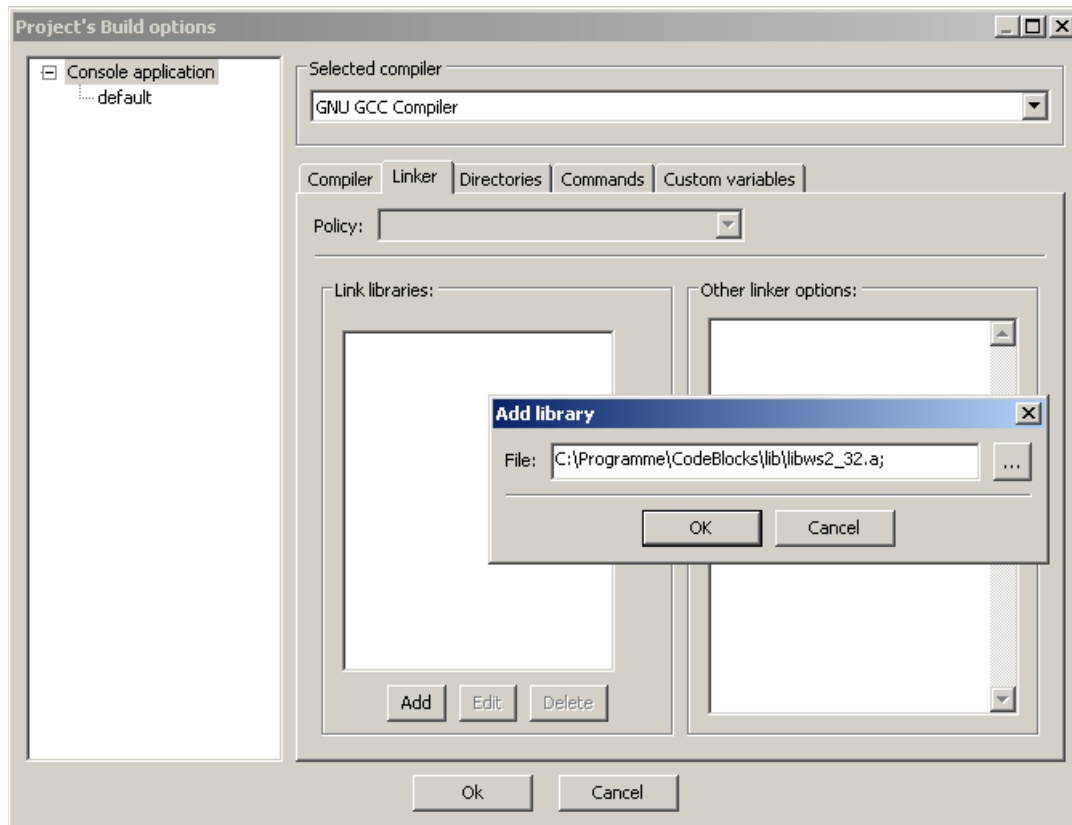


Abb. 1.3.2 Eingeben der Winsock-Lib.

Sollten Sie Nutzer der **MS VisualC++ IDE** sein, müssen Sie die `WS2_32.lib` Bibliothek dem Linker bekannt geben. Dazu gehen Sie unter Projekt->Eigenschaften->Linker und tragen, wie im Screenshot zu sehen, die Library unter "Zusätzliche Abhängigkeiten" ein:

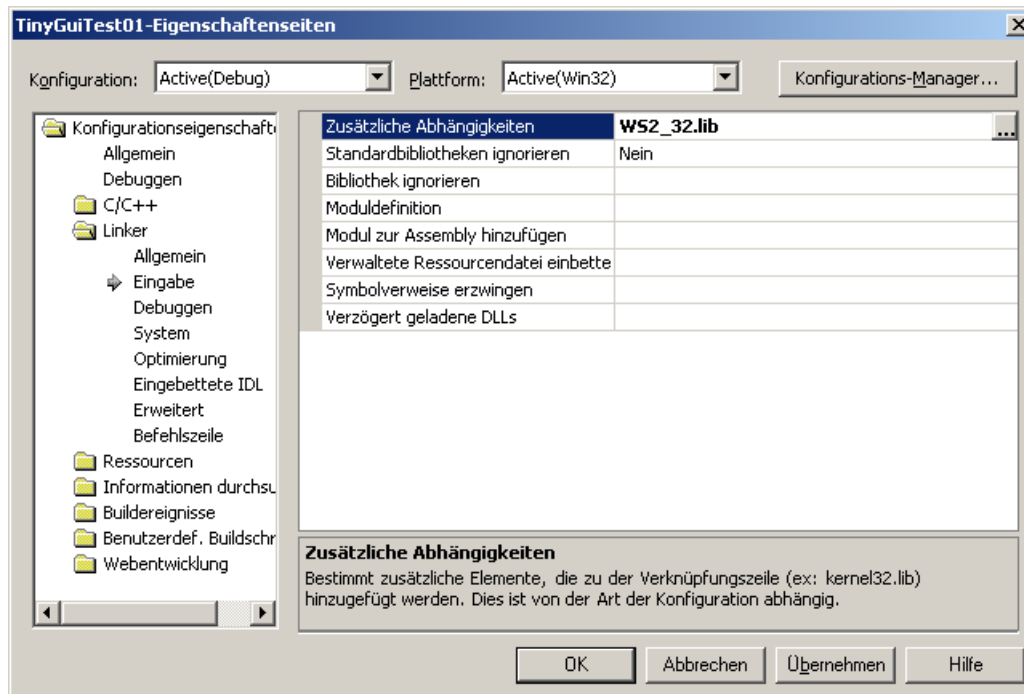


Abb. 1.3.3 Zusätzliche Abhängigkeiten

2 Das erste Socket-Programm

Unser erstes Programm soll einfach nur über den [Port](#) 80, welcher der standardmäßige Port für das HTTP-Protokoll ist, eine Verbindung aufbauen.
Zuerst müssen Sie die nötigen Headerdateien inkludieren:

```
C/C++ Code:
1 #include <iostream>
2 #ifdef linux
3 #include <sys/socket.h> // socket(), connect()
4 #include <arpa/inet.h> // sockaddr_in
5 #else
6 #include <winsock2.h>
7 #endif
8
9 int main()
10 {
11     using namespace std;
```

Die Konstante linux wird von meinem Compiler automatisch definiert, wenn ich unter Linux bin. Nähere Infos: <http://predef.sourceforge.net/preos.html>

Sie können auch einfach nur den für Ihr Betriebssystem benötigten Code nehmen und den Rest verwerfen.

Die iostream-Headerdatei ist klar; für Windows müssen wir nur die winsock2.h inkludieren, dort sind alle Funktionen für die zweite Winsock-Version definiert. Bei Linux sind die einzelnen Sachen in verschiedene Dateien aufgeteilt.

Folgender Code ist nur für Windows wichtig:

```
C/C++ Code:
1 #ifndef linux
2     WSADATA w;
3     if(int result = WSStartup(MAKEWORD(2,2), &w) != 0)
4     {
5         cout << "Winsock 2 konnte nicht gestartet werden! Error #" << result << endl;
6         return 1;
7     }
8 #endif
```

Mit der Funktion WSStartup wird Windows mitgeteilt, dass man gerne Zugriff auf die Winsock-Library haben will. Die Parameter sind eigentlich unwichtig, falls diese Sie dennoch interessieren, können Sie auf der [MSDN-Seite](#) nachschauen, was sie bedeuten. Das einzige was wir uns merken müssen, ist dass jedes Winsock-Programm mit dieser Funktion starten sollte, bevor es irgendwelche anderen Socket-Funktionen aufruft.

Nun geht es weiter in der main-Funktion:

```
C/C++ Code:
    int Socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if(Socket == -1)
    {
        cout << "Socket konnte nicht erstellt werden!" << endl;
        return 1;
    }
```

Die socket()-Funktion erstellt ein neues Socket, einen „Netzwerkanschluss“, und gibt dessen ID zurück. Mit dieser ID können wir später auf dem Socket Daten senden bzw. empfangen. Ein negativer Wert stellt hierbei einen Fehler beim Erstellen dar. Weitere Infos zu der socket-Funktion gibt es [hier](#).

Hinweis: Als Windows-User werden Sie wahrscheinlich auf den Datentyp SOCKET treffen. Dieser ist ein typedef auf unsigned int. Trotzdem können Sie ein Socket einfach als int behandeln, da es eine implizite Umwandlung zwischen diesen Typen gibt.

Nachdem wir ein Socket erstellt haben, müssen wir eine Verbindung aufbauen.

Jede Verbindung hat verschiedene Parameter, die in einer Struktur namens sockaddr gespeichert sind. Sie müssen sich das so vorstellen, dass die sockaddr eine abstrakte Basisklasse ist (die nie benutzt wird) und Strukturen wie sockaddr_in sind die abgeleiteten Klassen.

(Dass man nicht gleich ein Klassendesign gewählt hat, liegt daran, dass Sockets auch unter C funktionieren sollen)

Wir benötigen nur die sockaddr_in, sie ist für die normalen, vierstelligen [IP-Adressen](#) zuständig (z.B. 192.168.114.100). Falls Sie sich mit dem neuen [IPv6](#) beschäftigen wollen, benötigen Sie dann eine andere sockaddr-Struktur. Eine gute Übersicht gibt's [hier](#).

Der erste Parameter dieser Strukturen ist immer der Typ der sockaddr-Struktur:

```
C/C++ Code:
    sockaddr_in service; // Normale IPv4 Struktur
    service.sin_family = AF_INET; // AF_INET für IPv4, für IPv6 wäre es AF_INET6
```

Als nächstes müssen wir einen Port festlegen, auf dem wir connecten wollen, dies wäre beim HTTP-Protokoll der Port 80. Hierbei ist zu beachten, dass die Struktur den Port in umgekehrter Bytereihenfolge speichert. Also zuerst das eigentlich hintere Byte von short und dann das

vordere. Klingt komisch, ist aber eigentlich gar nicht so schwer, denn um eine normale Zahl in die umgekehrte Reihenfolge zu bringen, gibt es schon die Funktion `htons()`. Für genauere Infos suchen Sie im Internet nach [big-endian](#).

C/C++ Code:

```
service.sin_port = htons(80); // Das HTTP-Protokoll benutzt Port 80
```

Falls Sie noch Probleme haben, das mit der Bytereihenfolge zu verstehen, gibt es hier ein kleines Beispiel für eine mögliche Implementierung dieser Funktion:

C/C++ Code:

```
1 unsigned short my_htons(unsigned short h)
2 {
3     char* p = reinterpret_cast<char*>(&h);
4     char n[2];
5     n[0] = p[1];
6     n[1] = p[0];
7     return *reinterpret_cast<unsigned short*>(n);
8 }
```

Falls Sie später `htons` durch `my_htons` ersetzen, sollte es genauso funktionieren.

Jetzt kommen wir zum Interessanten: Der IP. Bei unserem ersten Programm soll der User erstmal nur eine IP eingeben, zu der dann verbunden wird.

C/C++ Code:

```
string ip;
cout << "IP: ";
cin >> ip;
```

Nun wird es wieder etwas schwieriger. Wie Sie wissen, wird die IP in der Form 123.44.32.99 dargestellt. Die einzelnen vier Werte gehen von 0 bis 255, und da muss jedem Programmierer sofort auffallen, dass dies ein `unsigned char` ist. Also haben wir 4 `unsigned chars` die durch einen Punkt getrennt sind. Dies ist aber nur eine für Menschen leserlich gemachte Form. In Wirklichkeit speichert der Computer natürlich keinen String sondern die 4 Bytes direkt. Also haben wir ein 4 Byte großes Array aus `unsigned char`. Klingt logisch; macht Sie das nicht misstrauisch? Richtig! Natürlich speichert man kein Array, sondern einen `unsigned long`. Wäre vielleicht ja noch ganz logisch, wenn man diesen anstelle einer Struktur benutzt, aber nein, den packen wir den nochmal in eine Struktur rein.

C/C++ Code:

```
struct in_addr
{
    unsigned long s_addr; // long ist 4 bytes also 4 chars groß
};
```

Dass dies sinnlos ist, haben sich die Entwickler bei Microsoft wohl auch gedacht. Also haben sie die `in_addr` bei Winsock neu entworfen. In deren Struktur gibt es vier `unsigned chars`, zwei `unsigned shorts` und einen `unsigned long`. Damit das ganze kompatibel ist, hat man diese Variablen in eine [union](#) gepackt. Somit lassen sich immernoch Casts von `unsigned long*` zu einem `in_addr*` durchführen, dazu aber später mehr. Wichtig ist: Vergessen Sie die Microsoft-Version der `in_addr` und stellen Sie sich einfach vor, die `in_addr`-Struktur enthält nur einen `unsigned long`, der die IP-Adresse in binärer Form speichert.

Da wir ja wollen, dass der User nicht die IP-Adresse binär eingibt, sondern in der gewohnten Form, müssen wir sie umwandeln. Zum Glück gibt es dafür auch schon eine Funktion namens `inet_addr`. Diese gibt einen `unsigned long` zurück, den wir dann einfach an den `s_addr`-Member der `in_addr`-Struktur übergeben. Dieser heißt in der `sockaddr_in`-Struktur `sin_addr` (Siehe Übersicht der `sockaddr_in`-Struktur).

C/C++ Code:

```
service.sin_addr.s_addr = inet_addr(ip.c_str());
```

Nun haben wir alle Informationen für eine Verbindung an die vom User eingegebene IP gespeichert. Es ist an der Zeit nun wirklich zu verbinden. Dazu gibt es die Funktion `connect`. Diese erwartet als ersten Parameter ein `Socket`, als zweiten einen Zeiger auf unsere `sockaddr`-Struktur, den wir natürlich casten müssen, da wir ja in Wirklichkeit eine `sockaddr_in` Struktur haben. Und als letztes folgt die Länge unserer `sockaddr`-Struktur. Die Länge muss übergeben werden, weil... äh.. ist eigentlich eine gute Frage, denn theoretisch, kann die `connect`-Funktion ja durch die `sin_family` herausfinden wie lang die bestimmte `sockaddr`-Struktur ist. Aber hier hat man sich anscheinend dazu entschieden, die Länge zur Sicherheit auch noch zu übergeben, vielleicht weil sie nicht plattformunabhängig ist.

C/C++ Code:

```
int result = connect(Socket, reinterpret_cast<sockaddr*>(&service), sizeof(service));
```

Der Rückgabewert von `connect` ist bei einem Fehler -1. Dies überprüfen wir und geben bekannt, falls die Verbindung fehlgeschlagen ist:

C/C++ Code:

```

if(result == -1)
{
    cout << "Verbindung fehlgeschlagen!" << endl;
    return 1;
}

cout << "Verbindung erfolgreich!" << endl;

```

An dieser Stelle haben wir jetzt eine Verbindung mit dem Server aufgebaut und könnten theoretisch Daten austauschen, das verschieben wir aber mal lieber auf das nächste Kapitel und beenden jetzt schon die Verbindung:

```

C/C++ Code:
#ifdef linux
    close(Socket);
#else
    closesocket(Socket);
#endif
}

```

Diese Funktion schließt das Socket, dessen Identifizierungsnummer hier übergeben wird. Unter Linux heißt sie close() und unter Windows closesocket(). Falls wir hier nach nochmal connect() oder eine andere Funktion aufrufen, die ein Socket erwartet, werden wir eine Fehlermeldung erhalten, denn diese ID zeigt nicht auf ein gültiges Socket. Wir beachten, dass die closesocket-Funktion threadsicher ist und alle blockierenden Socket-Funktionen, die dieses Socket benutzen, stoppt. Auch wird ein bestimmter Status gesendet, sodass das Gegenüber weiß, dass die Verbindung geschlossen wurde. Also schrecken Sie nicht davor zurück, close zu benutzen.

Unser Programm sollte nun [so](#) aussehen. Nun können wir es kompilieren und es sollte keine Fehlermeldung erscheinen. Testen Sie eine IP, hinter der Sie einen Webserver wissen. Falls Ihnen keine einfällt, können Sie z.B. mit dem Konsolen-Befehl

```

Code:
nslookup www.google.de

```

die IP-Adresse von Google erfahren. (Die Konsole starten Sie unter Windows mit [Windowstaste]+[R] und dann „cmd“ ausführen.) Die Verbindung sollte erfolgreich zustande kommen. Probieren Sie auch einfach irgendetwas beliebiges aus, denn nun sollte es eine Fehlermeldung geben. Eine Konsolenausgabe könnte z.B. so aussehen (Linux):

```

Code:
1 jhasse@jhasse-desktop:~/C++/http$ g++ 01.cpp
2 jhasse@jhasse-desktop:~/C++/http$ nslookup www.google.de
3 Server:      217.237.149.161
4 Address:     217.237.149.161#53
5
6 Non-authoritative answer:
7 www.google.de canonical name = www.google.com.
8 www.google.com canonical name = www.l.google.com.
9 Name: www.l.google.com
10 Address: 209.85.129.104
11 Name: www.l.google.com
12 Address: 209.85.129.147
13 Name: www.l.google.com
14 Address: 209.85.129.99
15
16 jhasse@jhasse-desktop:~/C++/http$ ./a.out
17 IP: 209.85.129.104
18 Verbindung erfolgreich!
19 jhasse@jhasse-desktop:~/C++/http$ ./a.out
20 IP: 209.85.129.147
21 Verbindung erfolgreich!
22 jhasse@jhasse-desktop:~/C++/http$ ./a.out
23 IP: 209.85.129.99
24 Verbindung erfolgreich!
25 jhasse@jhasse-desktop:~/C++/http$ ./a.out
26 IP: 123.123.123.123
27 Verbindung fehlgeschlagen!
28 jhasse@jhasse-desktop:~/C++/http$

```

Bis hierhin sollten Sie alles verstanden haben. Falls es dennoch Probleme gibt, schauen Sie noch mal in die Referenzen und anderen Tutorials (siehe Links am Ende).

3 Unser eigenes nslookup

Jetzt ist es natürlich sehr nervig, dass man immer eine IP eingeben muss. Es wäre doch viel praktischer, wenn man wie beim Browser einfach einen Namen eingibt und das Programm diesen automatisch auflöst. Also fangen wir an: Programmieren wir uns unser eigenes nslookup.

Die Funktion, die uns interessiert heißt `gethostbyname` und gibt einen Zeiger auf eine `hostent`-Struktur zurück. Diese wollen wir uns mal genauer anschauen:

```
C/C++ Code:
1 struct hostent
2 {
3     char* h_name; /* Offizieller Name des Host */
4     char** h_aliases; /* Weitere Namen für diesen Host */
5     int h_addrtype; /* Adresstyp, meistens AF_INET */
6     int h_length; /* Länge einer IP-Adresse in Bytes, meistens 4 */
7     char** h_addr_list; /* Die IP-Adressen */
8 };
```

Der erste Parameter ist ein C-Array, der den Namen des Host speichert. Dieser ist für uns später unwichtig genauso wie die weiteren Namen des Hosts. Da diese mehrere sein können, haben wir eine Liste aus Zeigern, bzw aus C-Arrays. Das wird jetzt sehr kompliziert, deswegen ein kurzes Beispiel:

```
Code:
char* = C-String
char** = C-String*
```

Wir haben also einen Zeiger auf einen C-String. Aber wozu einen Zeiger? Klar, weil es sich um ein dynamisches Array handelt:

```
Code:
Dynamisches Array in C:    T* p = (T)malloc(size);
Dynamisches Array in C++:  std::vector<T> v(size);
```

Wenn also in C ein dynamisches Array ein Zeiger ist, der auf einen Speicherbereich zeigt, dann ist es in C++ ein Vector. Also würde `h_aliases` in C++ so aussehen:

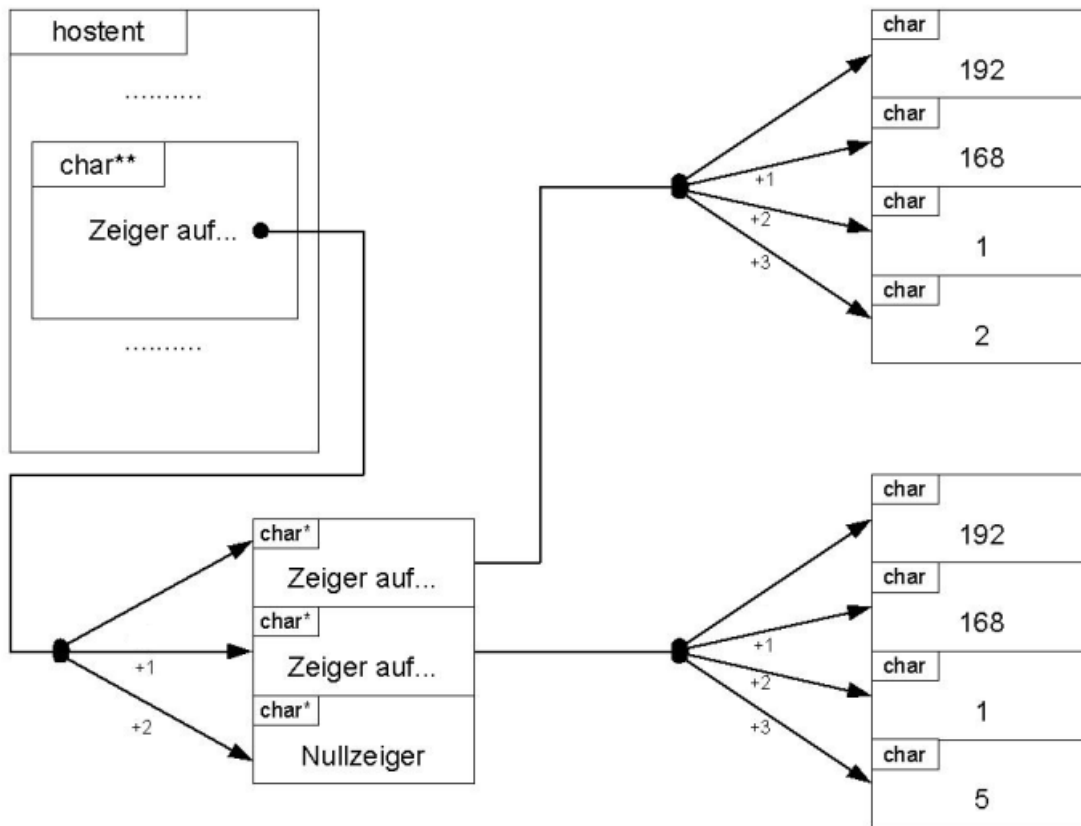
```
C/C++ Code:
std::vector<std::string> h_aliases;
```

Schade, dass die `hostent`-Struktur in C geschrieben wurde, aber ich hoffe, Sie haben das Prinzip verstanden.

Der nächste Typ beschreibt den Adresstyp, bei uns also einfach nur `AF_INET`, IPv6-Adressen sind uns egal. Also ist auch der nächste Parameter nicht relevant und sollte eigentlich immer 4 betragen.

Wichtig ist nun die Liste der IP-Adressen. Diese sieht am Anfang genau so aus wie die Liste der Aliases, doch passen Sie auf: es ist nicht das gleiche. Hier haben wir keine Strings, denn die IP-Adressen werden, wie in der `sockaddr`, binär gespeichert.

Da ein Bild mehr als 1000 Worte sagt, ist hier mal ein Bild:



Dies ist der grundsätzliche Aufbau der `hostent`-Struktur. Kommen wir zurück zur `gethostbyname`-Funktion. Diese erwartet einen C-String, in dem der Hostname in der Form www.bla.de enthalten ist. Unser neues Programm fängt also so an:

C/C++ Code:

```

1 #include <iostream>
2 #ifndef linux
3 #include <netdb.h> // gethostbyname(), hostent
4 #include <arpa/inet.h> // inet_ntoa()
5 #else
6 #include <winsock2.h>
7 #endif
8
9 int main()
10 {
11     using namespace std;
12
13     #ifndef linux
14         WSADATA w;
15         if(int result = WSStartup(MAKEWORD(2,2), &w) != 0)
16         {
17             cout << "Winsock 2 konnte nicht gestartet werden! Error #" << result << endl;
18             return 1;
19         }
20     #endif
21
22     cout << "Bitte gebe einen Hostnamen ein: ";
23     string Hostname;
24     cin >> Hostname;
25
26     hostent* phe = gethostbyname(Hostname.c_str());

```

Dass der Zeiger auf die `hostent`-Struktur wieder freigegeben wird, soll uns nicht kümmern, denn dies wird automatisch erledigt. Erstmal sollten wir checken, ob der Hostname überhaupt existiert:

C/C++ Code:

```

if(phe == NULL)
{
    cout << "Host konnte nicht aufgeloeset werden!" << endl;
    return 1;
}

```

Nun geben wir den Namen sowie die Aliases aus:

```

C/C++ Code:
1  cout << "hostname: " << phe->h_name << endl
2  << "Aliases: ";
3
4  for(char** p = phe->h_aliases; *p != 0; ++p)
5  {
6      cout << *p << " ";
7  }
8  cout << endl;

```

Die Funktion der for-Schleife ist nicht ganz einfach: p zeigt auf den ersten C-String und wird nach jedem Schleifendurchlauf um 1 erhöht, bis wir einen Nullzeiger haben. Wichtig: Keinen leeren String der ein '\0' enthält, sondern die Liste von Zeigern auf C-Strings enthält einen Nullzeiger, der nicht auf einen C-String zeigt.

Als nächstes wird einfach nur überprüft, ob es sich bei den IPs um IPv4-Adressen handelt. IPv6 lassen wir außer Acht.

```

C/C++ Code:
1  if(phe->h_addrtype != AF_INET)
2  {
3      cout << "Unqueltiger Adresstyp!" << endl;
4      return 1;
5  }
6
7  if(phe->h_length != 4)
8  {
9      cout << "Unqueltiger IP-Typ!" << endl;
10     return 1;
11 }

```

Nun wollen wir diese IP-Adressen ausgeben, doch sie liegen in binärer Form vor. Also müssen wir sie in einen String umwandeln und hierzu gibt es die Funktion `inet_ntoa()`. Sie wandelt eine `in_addr`-Struktur in einen String um, in der Form „x.x.x.x“. Leider haben wir keine Zeiger auf `in_addr`-Strukturen sondern Zeiger auf chars. Da die `in_addr`-Struktur die Daten auch nur binär speichert, können wir den Zeiger einfach in einen `in_addr`-Zeiger casten:

```

C/C++ Code:
reinterpret_cast<in_addr*>(*phe->h_addr_list);

```

Dieser neue Zeiger muss nun noch dereferenziert werden, da `inet_ntoa()` eine Instanz und keinen Zeiger erwartet:

```

C/C++ Code:
cout << inet_ntoa(*reinterpret_cast<in_addr*>(*phe->h_addr_list));

```

Nun müssen wir noch, wie bei den Aliases, die Liste wirklich durchgehen und nicht einfach nur das erste Element nehmen, da ein Host ja auch mehrere IPs haben kann. Der endgültige Code sieht also so aus:

```

C/C++ Code:
cout << "IP-Adressen: ";
for(char** p = phe->h_addr_list; *p != 0; ++p)
{
    cout << inet_ntoa(*reinterpret_cast<in_addr*>(*p)) << " ";
}
cout << endl;
}

```

Um die Funktion der `inet_ntoa`-Funktion etwas klarer zu machen, hier nochmal eine mögliche Implementierung:

```

C/C++ Code:

```



```

1 #include <sstream>
2 std::string my_inet_ntoa(in_addr& ip)
3 {
4     unsigned char* p = reinterpret_cast<unsigned char*>(&ip);
5     std::stringstream sstream;
6     for(int i = 0; i < 4 && (i == 0 || (sstream << "_")); ++i)
7     {
8         sstream << static_cast<int>(p[i]);
9     }
10    return sstream.str();
11 }

```

Nun sind wir schon fertig. Das [endgültiges Programm](#) könnt ihr nun testen, mit einer Adresse wie www.google.de oder www.microsoft.com (oder www.kernel.org :P). Alles sollte klappen und als nächstes wollen wir diesen Code in das vorherige Programm einfügen.

4 Integration von nslookup

Nun wollen wir unser eigenes nslookup in das Programm aus Kapitel 2 einfügen. Danach sollte der Benutzer nur noch den Namen der Internetseite eingeben müssen und das Programm kümmert sich um die Verbindung. Dies klingt einfacher als es ist, da ein Host ja mehrere IP-Adressen haben kann und diese müssen alle ausprobiert werden, bevor gesagt wird: "Keine Verbindung möglich!". Denn das wäre ja gelogen 😊.

Fangen wir zuerst an wie beim Programm vom vorherigen Kapitel:

C/C++ Code:

```

1 #include <iostream>
2 #ifdef linux
3 #include <netdb.h> // gethostbyname(), hostent
4 #include <arpa/inet.h> // inet_ntoa()
5 #else
6 #include <winsock2.h>
7 #endif
8
9 int main()
10 {
11     using namespace std;
12
13     #ifndef linux
14         WSADATA w;
15         if(int result = WSStartup(MAKEWORD(2,2), &w) != 0)
16         {
17             cout << "Winsock 2 konnte nicht gestartet werden! Error #" << result << endl;
18             return 1;
19         }
20     #endif
21
22     cout << "Bitte gebe einen Hostnamen ein: ";
23     string Hostname;
24     cin >> Hostname;
25
26     hostent* phe = gethostbyname(Hostname.c_str());
27
28     if(phe == NULL)
29     {
30         cout << "Host konnte nicht aufgelöst werden!" << endl;
31         return 1;
32     }
33
34     cout << "\nHostname: " << phe->h_name << endl;
35     << "Aliases: ";
36
37     for(char** p = phe->h_aliases; *p != 0; ++p)
38     {
39         cout << *p << " ";
40     }
41     cout << endl;
42
43     if(phe->h_addrtype != AF_INET)
44     {
45         cout << "Unqueltiger Adresstyp!" << endl;
46         return 1;
47     }
48
49     if(phe->h_length != 4)
50     {
51         cout << "Unqueltiger IP-Typ!" << endl;

```

```
52     return 1;
53 }
```

Nun müssen wir jede einzelne IP-Adresse testen, ob sie klappt. Dazu gehen wir die Liste vom Anfang durch und sobald wir eine Verbindung gefunden haben, fahren wir fort. Falls das Ende der Liste erreicht wurde und keine IP-Adresse klappte, brechen wir ab. Mit ein bisschen Logik kann man daraus eine Schleife erstellen, ich habe es so gemacht:

C/C++ Code:

```
1  int Socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
2  if(Socket == -1)
3  {
4      cout << "Socket konnte nicht erstellt werden!" << endl;
5      return 1;
6  }
7
8  sockaddr_in service;
9  service.sin_family = AF_INET;
10 service.sin_port = htons(80); // Das HTTP-Protokoll benutzt Port 80
11
12 char** p = phe->h_addr_list; // p mit erstem Listenelement initialisieren
13 int result; // Ergebnis von connect
14 do
15 {
16     if(*p == NULL) // Ende der Liste
17     {
18         cout << "Verbindung fehlgeschlagen!" << endl;
19         return 1;
20     }
21
22     service.sin_addr.s_addr = *reinterpret_cast<unsigned long*>(*p);
23     ++p;
24     result = connect(Socket, reinterpret_cast<sockaddr*>(&service), sizeof(service));
25 }
26 while(result == -1);
27
28 cout << "Verbindung erfolgreich!" << endl;
```

Ich erstelle eine Variable result, die das Ergebnis von connect speichert, eine Variable p die als Iterator dient. Als erstes prüfe ich, ob ein Nullzeiger vorliegt, also ob die Liste zu Ende ist. Wenn nicht, erstelle ich meine binäre IP-Adresse und erhöhe danach schonmal den p-Zeiger (irgendwo muss ich's ja tun). Jetzt wird versucht zu verbinden. Falls result -1 ist, also fehlgeschlagen, dann wird das Ganze wiederholt, wenn nicht geht's nach der Schleife weiter.

Nun müssen wir nur noch die Verbindung beenden und unser [Programm](#) ist fertig.

5 Senden und Empfangen

5.1 Grundlegender Aufbau des HTTP-Protokolls

Nun wollen wir uns endlich mit dem Senden und Empfangen von Daten beschäftigen. Was bei uns eine Internetadresse ist, besteht aus folgenden Abschnitten:

Code:

```
http://www.kernel.org/faq/index.html
1 | 2 | 3
```

Zuerst kommt das Protokoll (1), danach der Host den wir auflösen (2) und als letztes die Datei bzw. der Pfad, den wir an den Webserver schicken müssen. Die Kommunikation zwischen Browser und Webserver geschieht hierbei über das HTTP-Protokoll. Es handelt sich also um eine Sprache, mit der der Dateiaustausch über das Internet geregelt werden kann. Ein anderes Protokoll wäre z.B. das FTP-Protokoll. Nun müssen Sie wissen, wie das HTTP-Protokoll funktioniert: Ein Webserver horcht auf Port 80. Sobald sich ein Client verbindet, wird auf zu empfangende Daten gewartet. Der Client, meistens der Browser, in diesem Fall aber unser Programm, schickt nun eine Anfrage, welche Datei er haben möchte. Bei dieser Anfrage handelt sich um einfachen Text im [ASCII-Format](#) (Deswegen auch unter anderem die Probleme beim Realisieren von Domainnamen mit Umlauten). Eine Anfrage, die wir dem Server nach dem Verbindungsaufbau schicken, sieht so aus:

Code:

```
GET /faq/index.html HTTP/1.1
Host: www.kernel.org
(hier eine leere Zeile)
```

Zuerst kommt ein Befehl, in diesem Fall GET, gefolgt von einem Leerzeichen. Nun kommt die Datei, die wir anfordern, noch ein Leerzeichen und dann die Protokollversion (HTTP/1.0 wäre zum Beispiel die ältere Variante). Nun kommen wir in die nächste Zeile, dabei müssen wir aber eines beachten: Die [Zeilenumbrüche](#) sind hier im DOS-Format, das heißt wir haben zuerst ein \r-Zeichen (ASCII-Code 13) und ein \n-Zeichen (ASCII-Code 10). Wollen wir nun also einen String erstellen der unsere Anfrage enthält, würde es so aussehen:

C/C++ Code:

```
const string request = "GET /faq/index.html HTTP/1.1\r\nHost: www.kernel.org\r\n\r\n";
```

Am Ende haben wir ein `\r\n\r\n`, hierbei handelt es sich einfach um die leere Zeile, die am Ende gesendet werden muss, damit der Server erkennt, dass hier die Anfrage zu Ende ist.

Nach der Anfrage kommt natürlich die Antwort. Diese besitzt auch einen ganz bestimmten Aufbau, den wir uns aber erst im nächsten Kapitel anschauen wollen. Jetzt kommen wir zuerst zum allgemeinen Senden und Empfangen von Daten.

5.2 Die Funktionen send und recv

Das Senden bzw. Empfangen geschieht mit Hilfe folgender Funktionen:

C/C++ Code:

```
int send(int sockfd, const void *msg, int len, int flags);
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

Der Aufbau [beider Funktion](#) ist ziemlich gleich. Der erste Parameter ist der Socket, auf dem gesendet werden soll, der zweite ein Zeiger auf einen Binärspeicherbereich mit der Länge len. Der letzte Parameter flags ist für uns unwichtig und es wird einfach 0 übergeben.

Nicht zu vergessen ist der Rückgabewert: Hierbei handelt es sich um die Anzahl der übertragenen Bytes. Denn auch wenn man versucht, einen Buffer von 100 Byte zu übertragen, kann send damit nicht fertig werden und wird vorher schon aufhören und dann die geschaffte Anzahl zurückgeben, z.B. 34. Unsere Aufgabe ist es also den restlichen Buffer von 64 Bytes noch zu senden. Dazu schreiben wir eine Funktion, die uns diese Arbeit immer abnehmen soll:

C/C++ Code:

```
1 void SendAll(int socket, const char* const buf, const int size)
2 {
3     int bytesSent = 0; // Anzahl Bytes die wir bereits vom Buffer gesendet haben
4     do
5     {
6         bytesSent += send(socket, buf + bytesSent, size - bytesSent, 0);
7     } while(bytesSent < size);
8 }
```

Diese Funktion sendet den von uns übergebenen Buffer komplett. Doch was passiert nun, wenn während dem Senden ein Fehler auftritt?

Vielleicht bricht die Verbindung ab oder geht verloren. Dies signalisiert uns `send()`, indem es uns einen Wert kleiner 0 zurückgibt.

Da dies unerwartet ist, nehmen wir hierfür [Exceptions](#). Hierzu werden wir die [std::runtime_error-Klasse](#) verwenden. Jedes mal wenn wir sie werfen wollen, lassen wir sie von einer Funktion erstellen:

C/C++ Code:

```
std::runtime_error CreateSocketError()
{
```

Die Signatur der Funktion ist selbsterklärend, doch wenn wir zum Inhalt kommen, wird's kompliziert. Leider sind die Funktionen zum Erhalt von Fehlermeldungen unter Windows und Linux ziemlich verschieden. Alle Linux-Fans können jetzt aufatmen:

C/C++ Code:

```
std::ostringstream temp;
#ifdef linux
temp << "Socket-Fehler #" << errno << " " << strerror(errno);
```

Da der Konstruktor von `std::runtime_error` einen `std::string` erwartet, verwenden wir einen `std::ostringstream` zur einfachen Formatierung.

Wir geben eine Fehlernummer aus, gefolgt von einem Text, der den Fehler beschreibt. Natürlich müssen wir uns dies nicht selbst ausdenken.

Eine Fehlernummer wird in der globalen Variable `errno` gespeichert, mit der Funktion `strerror(int)` erzeugen wir einen Text den wir ausgeben können. Hierzu muss man nur noch die `errno.h` am Anfang inkludieren.

Unter Windows sieht das ganze so aus:

C/C++ Code:

```
1 #else
2 int error = WSAGetLastError();
3 temp << "Socket-Fehler #" << error;
4 char* msg;
5 if(FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
6                 NULL, error, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
7                 reinterpret_cast<char*>(&msg), 0, NULL))
8 {
9     try
10     {
11         temp << " " << msg;
```

```

12     LocalFree(msg);
13 }
14 catch(...)
15 {
16     LocalFree(msg);
17     throw;
18 }
19 }
20 #endif

```

Hier gibt die Funktion `WSAGetLastError()` eine Fehlernummer zurück und `FormatMessage` erzeugt daraus einen Text. Die genaue Funktionsweise lässt sich in der [MSDN](#) nachlesen.

Am Ende unserer Funktion erstellen wir eine Instanz der `std::runtime_error`-Klasse und geben sie zurück:

```

C/C++ Code:
    return std::runtime_error(temp.str());
}

```

Nun wollen wir von unserer neuen Funktion Gebrauch machen und fügen sie in die `SendAll`-Funktion ein:

```

C/C++ Code:
1 void SendAll(int socket, const char* const buf, const int size)
2 {
3     int bytesSent = 0; // Anzahl Bytes die wir bereits vom Buffer gesendet haben
4     do
5     {
6         int result = send(socket, buf + bytesSent, size - bytesSent, 0);
7         if(result < 0) // Wenn send einen Wert < 0 zurück gibt deutet dies auf einen Fehler hin.
8         {
9             throw CreateSocketError();
10        }
11        bytesSent += result;
12    } while(bytesSent < size);
13 }

```

Nun entwickeln wir noch eine Funktion für das Empfangen von Daten. Da das HTTP-Protokoll ja zeilenweise sendet, soll unsere Funktion auch so aufgebaut sein:

```

C/C++ Code:
// Liest eine Zeile des Sockets in einen stringstream
void GetLine(int socket, std::stringstream& line)
{

```

Wir lesen byteweise von dem Socket bis wir auf einen Zeilenumbruch treffen.

```

C/C++ Code:
    for(char c; recv(socket, &c, 1, 0) > 0; line << c)
    {
        if(c == '\n')
        {
            return;
        }
    }
}

```

Die Schleife wird nur verlassen, wenn `recv` einen Wert kleiner oder gleich 0 zurück gibt, somit werfen wir dahinter unsere Exception.

```

C/C++ Code:
    throw CreateSocketError();
}

```

Falls die `recv`-Funktion einen Wert gleich 0 zurück gibt, deutet dies auf einen normalen Verbindungsabbruch hin. Dies ist der Fall, wenn die andere Seite `close/closesocket` aufruft.

5.3 Request

Nun wollen wir ein Programm erstellen, das all diese Sachen anwendet. Wir schicken einen Request an www.kernel.org und geben dann erstmal einfach die Ausgabe auf der Konsole aus. Unser Programm fängt so an:

C/C++ Code:

```

1 #include <iostream>
2 #include <fstream>
3 #include <stdexcept> // runtime_error
4 #include <sstream>
5 #ifdef linux
6 #include <sys/socket.h> // socket(), connect()
7 #include <arpa/inet.h> // sockaddr_in
8 #include <netdb.h> // gethostbyname(), hostent
9 #include <errno.h> // errno
10 #else
11 #include <winsock2.h>
12 #endif
13
14 // Hier die Funktionen CreateSocketError, SendLine und GetLine einfügen
15
16 int main()
17 {
18     using namespace std;
19
20     #ifndef linux
21         WSADATA w;
22         if(int result = WSStartup(MAKEWORD(2,2), &w) != 0)
23         {
24             cout << "Winsock 2 konnte nicht gestartet werden! Error #" << result << endl;
25             return 1;
26         }
27     #endif
28
29     hostent* phe = gethostbyname("www.kernel.org");
30
31     if(phe == NULL)
32     {
33         cout << "Host konnte nicht aufgelöst werden!" << endl;
34         return 1;
35     }
36
37     cout << "\nHostname: " << phe->h_name << endl
38          << "Aliases: ";
39
40     for(char** p = phe->h_aliases; *p != 0; ++p)
41     {
42         cout << *p << " ";
43     }
44     cout << endl;
45
46     if(phe->h_addrtype != AF_INET)
47     {
48         cout << "Unqueltiger Adresstyp!" << endl;
49         return 1;
50     }
51
52     if(phe->h_length != 4)
53     {
54         cout << "Unqueltiger IP-Typ!" << endl;
55         return 1;
56     }
57
58     int Socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
59     if(Socket == -1)
60     {
61         cout << "Socket konnte nicht erstellt werden!" << endl;
62         return 1;
63     }
64
65     sockaddr_in service;
66     service.sin_family = AF_INET;
67     service.sin_port = htons(80); // Das HTTP-Protokoll benutzt Port 80
68
69     char** p = phe->h_addr_list; // p mit erstem Listenelement initialisieren
70     int result; // Ergebnis von connect
71     do
72     {
73         if(*p == NULL) // Ende der Liste
74         {
75             cout << "Verbindung fehlgeschlagen!" << endl;
76             return 1;
77         }
78

```

```

79     service.sin_addr.s_addr = *reinterpret_cast<unsigned long*>(&p);
80     ++p;
81     result = connect(Socket, reinterpret_cast<sockaddr*>(&service), sizeof(service));
82 }
83 while(result == -1);
84
85 cout << "Verbindung erfolgreich!" << endl;

```

Wie Sie sehen, nehmen wir als Hostnamen www.kernel.org, denn die Seite www.kernel.org/faq/index.html soll unsere Testseite werden. Nun wollen wir unsere Anfrage senden:

C/C++ Code:

```

const string request = "GET /faq/index.html HTTP/1.1\r\nHost: www.kernel.org\r\nConnection: close\r\n\r\n";

SendAll(Socket, request.c_str(), request.size());

```

Dies ist, bis auf das Hinzufügen der folgenden Zeile, die gleiche Anfrage:

Code:

```
Connection: close
```

Sie bewirkt, dass nachdem wir die Antwort erhalten haben, der Server die Verbindung schließt. Die Verbindung könnte z.B. aufrecht erhalten bleiben, wenn wir weitere Request hinterherschicken wollten, um z.B. Bilder zu laden.

Da das Behandeln der Response im nächsten Kapitel behandelt wird, wollen wir die Antwort vorläufig in eine Textdatei schreiben:

C/C++ Code:

```

1  ofstream fout("output.txt");
2
3  cout << "Empfange und schreibe Antwort in output.txt..." << endl;
4  while(true)
5  {
6      stringstream line;
7      try
8      {
9          GetLine(Socket, line);
10     }
11     catch(exception& e) // Ein Fehler oder Verbindungsabbruch
12     {
13         break; // Schleife verlassen
14     }
15     fout << line.str() << endl; // Zeile in die Datei schreiben.
16 }

```

Am Ende schließen wir wie gewohnt das Socket und unser Programm sieht [so](#) aus.

Wenn wir es ausführen, wird die Antwort die Response vom Server in die output.txt-Datei geschrieben. Diese sollte ungefähr so anfangen:

Code:

```

1 HTTP/1.1 200 OK
2 Date: Fri, 15 Dec 2006 14:51:19 GMT
3 Server: Apache/2.2.2 (Fedora)
4 Last-Modified: Wed, 08 Nov 2006 22:06:19 GMT
5 ETag: "b643c8-6118-421c3874aacc0"
6 Accept-Ranges: bytes
7 Content-Length: 24856
8 Connection: close
9 Content-Type: text/html
10 X-Pad: avoid browser bug
11
12 <?xml version="1.0" encoding="utf-8"?>
13 ...

```

6 Response

6.1 Statuszeile

Die erste Zeile beschreibt einen Statuscode. Sie besteht aus der HTTP-Version des Servers, einer Status-Nr. und einem Text, der diese beschreibt. Eine Liste aller Codes gibt's unter anderem bei [Wikipedia](http://de.wikipedia.org/wiki/Liste_der_HTTP-Statuscodes). Für uns wichtig ist zuerst einmal der Code 200: Er steht dafür, dass alles geklappt hat, unsere Anfrage in Ordnung war und die Datei existiert. Ein weiterer wichtiger Status-Code ist 100. Er sagt uns, dass der Server noch etwas Zeit braucht die Anfrage zu verarbeiten und uns gleich noch eine Antwort schicken wird. Das könnte z.B. so aussehen:

Code:

HTTP/1.1 100 Continue

HTTP/1.1 200 OK

Date: Fri, 15 Dec 2006 14:51:19 GMT

Server: Apache/2.2.2 (Fedora)

...

Nach der ersten Antwort die uns nur sagt, dass es gleich weiter geht, kommt eine freie Zeile und dann die richtige Antwort. In unserem letzten Programm machen wir jetzt nach dem Verbinden so weiter:

C/C++ Code:

```

1  const string request = "GET /faq/index.html HTTP/1.1\r\nHost: www.kernel.org\r\nConnection: close\r\n\r\n";
2  try
3  {
4      SendAll(Socket, request.c_str(), request.size());
5
6      int code = 100; // 100 = Continue
7      string Protokoll;
8      stringstream firstLine; // Die erste Linie ist anders aufgebaut als der Rest
9      while(code == 100)
10     {
11         GetLine(Socket, firstLine);
12         firstLine >> Protokoll;
13         firstLine >> code;
14         if(code == 100)
15         {
16             GetLine(Socket, firstLine); // Leere Zeile nach Continue ignorieren
17         }
18     }
19     cout << "Protokoll: " << Protokoll << endl;

```

Mit der GetLine-Funktion (siehe oben) lesen wir die erste Zeile in einen stringstream ein. Nun schreiben wir den Protokolltyp in einen String. Zur Erinnerung: Der >>-operator von streams liest bis zu einem Leerzeichen ein. Als nächstes lesen wir den Statuscode aus, wenn dieser 100 ist, müssen wir noch die leere Zeile zwischen der neuen Anfrage ignorieren und beginnen dann wieder von vorne. Nun müssen wir nur noch eine Fehlermeldung ausgeben, wenn ein Statuscode anders als 200 oder 100 auftrat:

C/C++ Code:

```

1  if(code != 200)
2  {
3      firstLine.ignore(); // Leerzeichen nach dem Statuscode ignorieren
4      string msg;
5      getline(firstLine, msg);
6      cout << "Error #" << code << " - " << msg << endl;
7      return 0;
8  }

```

6.2 Transfer-Arten

Nun kommen wir zum eigentlichen Header. Er besteht aus vielen Argumenten, von denen einige aber nebensächlich sind. Wichtig für uns ist erstmal die allgemeine Übertragungsart:

Normales Übertragen, Dateigröße nicht mit übergeben

Dies ist die einfachste Art der Übertragung. Sobald der Header zu Ende ist, lesen wir mit `recv` solange, bis der Server die Verbindung schließt. Wir erinnern uns: `recv` gibt beim Schließen der Verbindung 0 zurück.

Normales Übertragen, Dateigröße übergeben

Nun machen wir alles genauso wie vorher, nur das wir nicht darauf warten müssen, dass `recv` 0 zurück gibt: Wir hören einfach auf, sobald wir die Datei vollständig erhalten haben. Vorteil: Wir können den Fortschritt in % anzeigen.

Chunked-Encoding

Chunked-Encoding ist die komplizierteste Art: Die Datei wird nicht als ganzes übertragen, sondern in verschiedenen Abschnitten. Dies ist z.B. notwendig, wenn der Server ein PHP-Skript ausführt noch während die Seite gesendet wird.

Die für diese 3 Übertragungstypen wichtigen Argumente sind folgende:

Code:

Content-Length: 1234

Transfer-Encoding: chunked

Content-Length gibt uns die Größe der Datei in Bytes und das Argument Transfer-Encoding existiert nur dann, wenn auch das Chunked-Encoding angewendet wird.

Wir erstellen uns also eine bool-Variable, die speichert ob das "Transfer-Encoding: chunked" angegeben wurde, sowie eine Variable, die die übergebene Dateigröße speichert und mit einer Konstanten belegt wird, falls keine Größe angegeben wurde:

C/C++ Code:

```
bool chunked = false;
const int noSizeGiven = -1;
int size = noSizeGiven;
```

Nun starten wir mit der zeilenweisen Auslesung des Headers:

C/C++ Code:

```
1 while(true)
2 {
3     stringstream sstream;
4     GetLine(Socket, sstream);
5     if(sstream.str() == "\r") // Header zu Ende?
6     {
7         break;
8     }
```

Zur Erinnerung: Der Header endet mit einer leeren Zeile, da als Zeilenumbruch allerdings nicht \n verwendet wird (sondern \r\n), handelt es sich um eine Zeile, die nur ein \r enthält:

Code:

```
HTTP/1.1 200 OK\r\nAsd: 123\r\n\r\n
```

Nun lesen wir in einen String ein, der der Name des Arguments ist. Danach ignorieren wir ein Zeichen (das Leerzeichen), um danach den Wert einzulesen.

C/C++ Code:

```
string left; // Das was links steht
sstream >> left;
sstream.ignore(); // ignoriert Leerzeichen
```

Wenn der Name "Content-Size" ist, lesen wir die Größe ein:

C/C++ Code:

```
if(left == "Content-Length:")
{
    sstream >> size;
}
```

Falls wir auf eine Angabe über das Transfer-Encoding treffen, werten wir den Wert aus. Ist dieser "chunked", setzen wir die Bool-Variable:

C/C++ Code:

```
1 if(left == "Transfer-Encoding:")
2 {
3     string transferEncoding;
4     sstream >> transferEncoding;
5     if(transferEncoding == "chunked")
6     {
7         chunked = true;
8     }
9 }
10 }
```

Nun sind wir schon fertig mit der Auswertung des Headers. 😊

6.3 Die Übertragung

Nun beginnen wir mit der eigentlichen Übertragung der Datei. Dazu erstellen wir erstmal eine Output-Datei, in die wir binär schreiben werden:

C/C++ Code:


```
fstream fout("faq.html", ios::binary | ios::out);
if(!fout)
{
    cout << "Could Not Create File!" << endl;
    return 1;
}
```

Wir erstellen uns drei Variablen:

C/C++ Code:

```
int recvSize = 0; // Empfangene Bytes insgesamt
char buf[1024];
int bytesRecv = -1; // Empfangene Bytes des letzten recv
```

In der ersten Variable speichern wir, wieviel Bytes wir schon empfangen haben (dies ist z.B. für eine Prozentanzeige notwendig). Außerdem erstellen wir ein Array aus 1024 bytes für die Pufferung der Daten. Als letztes erstellen wir die temporäre Variable bytesRecv, die den letzten Rückgabe wert von recv speichert.

C/C++ Code:

```
if(size != noSizeGiven) // Wenn die Größe über Content-length gegeben wurde
{
    cout << "0%";
}
```

Wenn eine Größe übergeben wurde, können wir eine Fortschrittsanzeige ausgeben. Wir empfangen so lange, bis die gesamte Datei übertragen wurde:

C/C++ Code:

```
while(recvSize < size)
{
```

Nun schreiben wir die Daten, die wir auf unserem Socket empfangen, in den Buffer:

C/C++ Code:

```
if((bytesRecv = recv(Socket, buf, sizeof(buf), 0)) <= 0)
{
    throw CreateSocketError();
}
```

Falls recv einen Wert kleiner oder gleich 0 zurück gibt, brechen wir ab und werfen eine Exception. Nun addieren wir die aktuell empfangene Anzahl an Bytes zu der Gesamtanzahl und schreiben den Buffer in die Datei:

C/C++ Code:

```
recvSize += bytesRecv;
fout.write(buf, bytesRecv);
```

Als letztes geben wir noch eine Prozentanzeige aus und sind auch schon fertig:

C/C++ Code:

```
    cout << "\r" << recvSize * 100 / size << "%" << flush; // Mit \r springen wir an den Anfang der Zeile
}
}
```

Nun müssen wir noch die anderen zwei Transfer-Arten implementieren:

C/C++ Code:

```
1     else
2     {
3         if(!chunked)
4         {
5             cout << "Downloading... (Unknown Filesize)" << endl;
6             while(bytesRecv != 0) // Wenn recv 0 zurück gibt, wurde die Verbindung beendet
7             {
8                 if((bytesRecv = recv(Socket, buf, sizeof(buf), 0)) < 0)
9                 {
10                    throw CreateSocketError();
11                }
12            }
13        }
14    }
```

```

12         fout.write(buf, bytesRecv);
13     }
14 }

```

Dies ist die normale Übertragung ohne Angabe von Dateigröße. Der Unterschied besteht darin, dass wir nicht wissen, wann die Übertragung zu Ende ist, sondern einfach auf einen Verbindungsabbruch warten müssen.

6.4 Chunked Transfer-Encoding

Das letzte was uns nun noch fehlt, ist die Übertragung mit Chunks. Dazu müssen wir uns wieder den Aufbau angucken:

Code:

```

1 HTTP/1.1 200 OK
2 Transfer-Encoding: chunked
3
4 <Größe des Chunks in Hex>
5 <Daten des Chunks>
6 <Größe des zweiten Chunks in Hex>
7 <Daten des zweiten Chunks>
8 0
9 Weiteres Argument: Blabla
10 Noch ein Argument: 123
11 (hier eine leere Zeile)

```

Anstelle der Daten folgt bei dieser Übertragungs-Art eine Zeile in [hexadezimalen Schreibweise](#), der wir die Größe des Chunks in Bytes entnehmen können. Daraufhin kommen die Daten, gefolgt von einem Zeilenumbruch. Nun beginnt entweder ein neuer Chunk, oder es wird die Größe 0 übergeben, was für das Ende der Chunks steht. Hiernach können noch weitere Argumente folgen, die für uns aber unwichtig sind. Hauptsache wir haben die Datei und dann schnell weg :P.

Machen wir also weiter im Code, wo wir vorhin aufgehört haben:

C/C++ Code:

```

1     else
2     {
3         cout << "Downloading... (Chunked)" << endl;
4         while(true)
5         {
6             stringstream sstream;
7             GetLine(Socket, sstream);
8             int chunkSize = -1;
9             sstream >> hex >> chunkSize; // Größe des nächsten Parts einlesen

```

Wir lesen die erste Zeile ein und schreiben die hexadezimale Zahl in size. Hierzu brauchen wir nur das "hex"-Flag setzen, damit der Stream auch das Hexadezimalsystem anwendet.

Wenn die Größe kleiner oder gleich 0 ist, verlassen wir unsere Schleife:

C/C++ Code:

```

1     if(chunkSize <= 0)
2     {
3         break;
4     }

```

Nun beginnen wir eine Schleife die den aktuellen Chunk runterlädt:

C/C++ Code:

```

1     cout << "Downloading Part (" << chunkSize << " Bytes)..." << endl;
2     recvSize = 0; // Vor jeder Schleife wieder auf 0 setzen
3     while(recvSize < chunkSize)
4     {

```

Nun erstellen wir uns eine Variable, die nur speichert, wie viel Bytes wir diesen Schleifendurchgang noch empfangen müssen:

C/C++ Code:

```

1     int bytesToRecv = chunkSize - recvSize;

```

Dies ist nämlich notwendig, da wir ja am Ende unseres Chunks, nicht mehr empfangen dürfen als notwendig, sonst könnten wir z.B. versehentlich schon die Größe des nächsten Chunks in unsere Datei schreiben und das wäre fatal. Deswegen übergeben wir an recv als Größe entweder sizeof(buf) oder aber bytesToRecv, wenn wir nur noch ein kleines Stück empfangen müssen:

C/C++ Code:

```

    if(bytesRecv = recv(Socket, buf, bytesToRecv > sizeof(buf) ? sizeof(buf) : bytesToRecv, 0)) <= 0)
    {
        throw CreateSocketError();
    }

```

Nun addieren wir wie gewohnt die empfangenen Bytes und geben eine Prozentanzeige aus:

C/C++ Code:

```

    recvSize += bytesRecv;
    fout.write(buf, bytesRecv);
    cout << "\r" << recvSize * 100 / chunkSize << "%" << flush;
}
cout << endl;

```

Nun haben wir alles empfangen, doch halt: Haben wir nicht noch etwas vergessen? Richtig: Nach den Daten erfolgt ein Zeilenumbruch. Dieser besteht aus einem \r und einem \n, also 2 Bytes. Es ist also am einfachsten wenn wir eben 2 Bytes vom Socket ignorieren:

C/C++ Code:

```

    for(int i = 0; i < 2; ++i)
    {
        char temp;
        recv(Socket, &temp, 1, 0);
    }

```

Nun geben wir noch ein "Finished!" aus und unser Programm ist fertig:

C/C++ Code:

```

1      }
2      }
3      }
4      cout << endl << "Finished!" << endl;
5      }
6      catch(exception& e)
7      {
8          cout << endl;
9          cerr << e.what() << endl;
10     }
11     #ifndef linux
12         close(Socket); // Verbindung beenden
13     #else
14         closesocket(Socket); // Windows-Variante
15     #endif
16 }

```

Das [ganze Programm](#) sollte uns jetzt die `faq.html`-Datei herunterladen und auf der Festplatte speichern. Nun sind wir schon fast fertig.

7 Der letzte Feinschliff

Als letztes wollen wir unser Programm um ein paar Funktionen ergänzen, die eine Eingabe einer [URL](#) ermöglichen. Schließlich will man nicht immer nur eine Datei herunterladen. Dazu fügen wir am Anfang eine Eingabe hinzu:

C/C++ Code:

```

int main()
{
    using namespace std;

    cout << "URL: ";
    string URL;
    cin >> URL; // User gibt URL der Datei ein, die heruntergeladen werden soll
}

```

Nun müssen wir ein vor der URL evtl. existierendes [http://](#) entfernen:

C/C++ Code:

```

1 // Entfernt das http:// vor der URL
2 void RemoveHttp(std::string& URL)
3 {
4     size_t pos = URL.find("http://");
5     if(pos != std::string::npos)

```

```

6   {
7       URL.erase(0, 7);
8   }
9 }

```

Diese Funktion rufen wir nun nach dem Start von WinSock auf:

C/C++ Code:

```

1  #ifndef linux
2      WSADATA w;
3      if(int result = WSStartup(MAKEWORD(2,2), &w) != 0)
4      {
5          cout << "Winsock 2 konnte nicht gestartet werden! Error #" << result << endl;
6          return 1;
7      }
8  #endif
9
10 RemoveHttp(URL);

```

Jetzt extrahieren wir den Hostnamen aus der URL und speichern ihn in einem neuen String ab:

C/C++ Code:

```

string hostname = RemoveHostname(URL);

```

Die RemoveHostname-Funktion sieht so aus:

C/C++ Code:

```

1 // Gibt den Hostnamen zurück und entfernt ihn aus der URL, sodass nur noch der Pfad übrigbleibt
2 std::string RemoveHostname(std::string& URL)
3 {
4     size_t pos = URL.find("/");
5     if(pos == std::string::npos)
6     {
7         std::string temp = URL;
8         URL = "/";
9         return temp;
10    }
11    std::string temp = URL.substr(0, pos);
12    URL.erase(0, pos);
13    return temp;
14 }

```

Nun lösen wir den Hostnamen in dem neuen String auf:

C/C++ Code:

```

hostent* phe = gethostbyname(hostname.c_str());

```

Es folgt der Aufbau der Verbindung:

C/C++ Code:

```

if(phe == NULL)
// ...
cout << "Verbindung erfolgreich!" << endl;

```

Als nächstes müssen wir eine HTTP-Request erstellen:

C/C++ Code:

```

string request = "GET ";
request += URL; // z.B. /faq/index.html
request += " HTTP/1.1\n";
request += "Host: " + hostname + "\nConnection: close\n\n";

```

Diesen übergeben wir jetzt an unsere SendAll-Funktion und danach folgt der Code aus 05.cpp bis zur Öffnung des Filestreams:

C/C++ Code:

```

try
{
    SendAll(Socket, request.c_str(), request.size()); // Anfrage an Server senden
    int code = 100; // 100 = Continue
    // ...
}

```

Nun muss die Datei erstellt und der Inhalt eingelesen werden. Das größte Problem ist hierbei der Dateiname: Bei vielen Fällen lässt er sich extrahieren aber manchmal ist er auch unbekannt (z.B. www.google.de). Die sauberste Methode wäre es, das Response-Argument "Content-Type:" auszuwerten, dies wollen wir uns aber mal sparen und nennen unsere Datei einfach download und hängen eine Endung an, falls diese in der URL enthalten ist:

C/C++ Code:

```

1 // Gibt die Dateiendung in der URL zurück
2 std::string GetFileEnding(std::string& URL)
3 {
4     using namespace std;
5     size_t pos = URL.rfind(".");
6     if(pos == string::npos)
7     {
8         return "";
9     }
10    URL.erase(0, pos);
11    string ending = ".";
12    // Algorithmus um Sachen wie ?index=home nicht zuzulassen
13    for(string::iterator it = URL.begin() + 1; it != URL.end(); ++it)
14    {
15        if(isalpha(*it))
16        {
17            ending += *it;
18        }
19        else
20        {
21            break;
22        }
23    }
24    return ending;
25 }

```

C/C++ Code:

```

1 string filename = "download" + GetFileEnding(URL);
2 cout << "Filename: " << filename << endl;
3 fstream fout(filename.c_str(), ios::binary | ios::out);
4 if(!fout)
5 {
6     cout << "Could Not Create File!" << endl;
7     return 1;
8 }
9 int recvSize = 0; // Empfangene Bytes insgesamt
10
11 // ...
12 }
13 }
14 catch(exception& e)
15 {
16     cout << endl;
17     cerr << e.what() << endl;
18 }
19 #ifdef linux
20 close(Socket); // Verbindung beenden
21 #else
22 closesocket(Socket); // Windows-Variante
23 #endif
24 }

```

Fertig! Das komplette Programm können Sie sich [hier](#) herunterladen.

8 Nachwort

Falls Sie eine URL finden sollten, die dieses Programm nicht herunterladen kann, können Sie auf mein Tutorial antworten und mir den Link mitteilen, damit ich es mir ansehen kann.

Bei allgemeinen Fragen zum Thema Sockets können Sie in der [WinAPI](#)- oder der [Linux/Unix](#)-Abteilung nachfragen.

In meinem Code rufe ich recv auf, um nur ein einziges Byte zu empfangen. Dies sollte man in der Praxis möglichst unterlassen und immer

eine angemessene Buffergröße wählen, z.B. 4 KB. Hierdurch kann recv uns einzelne [TCP](#)-Pakete im Ganzen zurück geben. Um trotzdem zeilenweise einzulesen muss man einfach die recv-Funktion in seiner eigenen Funktion kapseln. Dies sollte man sowieso tun, da recv und send sehr low-level sind. Ein auf diese Weise verändertes Programm sähe [so](#) aus. Desweiteren wurde die gethostbyname-Funktion von der flexibleren getaddrinfo-Funktion ersetzt. Eine Erklärung sowie Beispiele finden Sie [hier](#).

Ich bedanke mich bei jcmds, flammenvogel, scrub, GPC, Artchi, predator und allen anderen aus der Redaktion, die mir geholfen haben. Das war's von mir, vielen Dank fürs Lesen!

9 Linkliste

Socket-Referenz

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/wsastartup_2.asp

Socket-Tutorial

<http://beej.us/guide/bqnet/>

HTTP-Tutorial

<http://www.jmarshall.com/easy/http/>

RFC für HTTP/1.1


<http://tools.ietf.org/html/rfc2616>

ten Mitglied 11:06:34 10.01.2007 Titel: **Re: Sockets und das HTTP-Protokoll**

joomoo schrieb:

C/C++ Code:

```
1 unsigned short my_htons(unsigned short h)
2 {
3     char* p = reinterpret_cast<char*>(&h);
4     char n[2];
5     n[0] = p[1];
6     n[1] = p[0];
7     return *reinterpret_cast<unsigned short*>(n);
8 }
```

so'n umständlichen endiantauscher hab' ich noch nie gesehen 
mach so:

C# Code:

```
unsigned short my_htons (unsigned short h)
{
    return (h>>8) | (h<<8);
}
```

oder so:

Code:

```
#ifdef BIG_ENDIAN
#define MY_HTONS(h) (h)
#else
#define MY_HTONS(h) ((h>>8)|(h<<8))
#endif
```

:xmas2:

addr Unregistrierter 11:54:14 10.01.2007 Titel:

gethostbyname sollte man nicht mehr benutzen, sondern man sollte getaddrinfo verwenden.

addr Unregistrierter 11:55:49 10.01.2007 Titel:

http://www.c-plusplus.de/magazin/bilder/sockets_und_das_http_protokoll/07.cpp

Der Link funktioniert nicht.

addr Unregistrierter 12:05:16 10.01.2007 Titel:

Zitat:

Dies signalisiert uns send(), indem es uns einen Wert gleich 0, für eine normale Beendigung der Verbindung

Das ist IMHO falsch. Wenn die Verbindung venünftig beendet wird kriegt man das nur über recv mit.

Zu WSASStartup: Wenn das fehlschlägt darf man WSAGetLastError nicht benutzen, sondern man muss den Rückgabewert als Fehlercode nehmen.

joomoo Autor 17:28:29 10.01.2007 Titel:

Hallo addr,

Vielen Dank für die Hinweise, die Datei hab ich gerade eben hochgeladen.
Die anderen Sachen schau ich mir auch eben an und antworte gleich nochmal (bin gerade nach Hause gekommen).

mfg.

joomoo Autor 17:34:45 10.01.2007 Titel: **Re: Sockets und das HTTP-Protokoll**

ten schrieb:**joomoo schrieb:****C/C++ Code:**

```
1 unsigned short my_htons(unsigned short h)
2 {
3     char* p = reinterpret_cast<char*>(&h);
4     char n[2];
5     n[0] = p[1];
6     n[1] = p[0];
7     return *reinterpret_cast<unsigned short*>(n);
8 }
```

so'n umständlichen endiantauscher hab' ich noch nie gesehen 🤔
mach so:

C# Code:

```
unsigned short my_htons (unsigned short h)
{
    return (h>>8) | (h<<8);
}
```

Klar ist der Endiantauscher umständlich, doch ich hab ihn ja auch nicht zum Einsetzen geschrieben sondern um die Funktionsweise zu demonstrieren. Deswegen wollte ich lieber auf Bitoperatoren verzichten.

mfg.

edit: Dies war ja gar nicht addr, sondern ten. Auch an dich ein großes Dankeschön!

joomoo Autor 19:19:14 10.01.2007 Titel:

Zitat:

gethostbyname sollte man nicht mehr benutzen, sondern man sollte getaddrinfo verwenden.

Oh, das wusste ich gar nicht. Ich hatte mich nach Beej's Network Guide gerichtet. Ändern kann ich es leider nicht mehr aber ich habe eine Bemerkung am Ende ergänzt.

Zitat:

Das ist IMHO falsch. Wenn die Verbindung venünftig beendet wird kriegt man das nur über recv mit.

Tatsächlich. Vielen Dank, ich hab es jetzt im Artikel korrigiert.

Zitat:

Zu WSAShutdown: Wenn das fehlschlägt darf man WSAGetLastError nicht benutzen, sondern man muss den Rückgabewert als Fehlercode nehmen.

Danke, ich hab's jetzt geändert und gebe den Rückgabewert aus.

mfg.

Tim Mitglied 11:14:12 11.01.2007 Titel:

Das eingescannte Bild ist einfach nur cool 😁👍

// Unregistrierter 11:31:38 11.01.2007 Titel:

Das eingescannte Bild wirkt einfach nur billig.