Module u. Bibliotheken

Bedeutung :

Umfangreiche Programmaufgaben können nur über eine Zerlegung in Teilaufgaben entwickelt werden.

Die **Zerlegung in Teilaufgaben** hat folgende Vorteile :

- Übersichtlichkeit und einfacheres Austesten der Programmteile
- Aufteilung der Programmierung auf mehrere Programmierer
- Erstellung und Verwendung von Bibliotheken für Standardaufgaben

Die Zerlegung muss dabei so vorgenommen werden, dass die Programmbausteine problemlos und universell in andere Programme eingebunden werden können.

Begriffe :

- **Modul** = Baustein, der universell und mehrfach verwendet werden kann
- **Bibliothek** = Sammlung von Bausteinen (Modulen)

Module müssen nach folgenden Gesichtspunkten erstellt werden :

- Datenübergabe über eine Schnittstelle (Parameter)
- Logische Zerlegung nach dem Gesichtspunkt eines Baukasten Systems - durch Kombination von Bausteinen sollen möglichst viele Aufgaben erledigt werden können
- Speicherung als fertig getestete Programmdateien

Modularisierung :

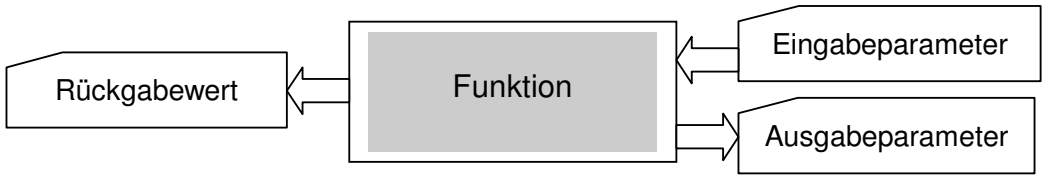
In C/C++ können Module entweder prozedural oder objektorientiert erstellt werden.

- prozedural = Unterprogramme (Funktionen) als Module
- objektorientiert = Klassen als Module

Unterprogramm als Modul :

Damit eine Funktion als Modul universell eingesetzt werden kann, muss die Datenübergabe an die Funktion über eine definierte Schnittstelle (Ein-Ausgabe-Parameter u. Rückgabewert) erfolgen.

Schema der Schnittstelle einer Funktion :



Funktions-Definition in C/C++ :

```
ReturnTyp   FunktionsName  ( formale Parameter )
{  lokale Definitionen
   Anweisungen
}
```

Über die Parameter-Schnittstelle können Daten an eine Funktion (Eingabeparameter) oder auch Daten aus einer Funktion (Ausgabeparameter u. Rückgabewert) flexibel übergeben werden.
Eine Funktion als universell einsetzbarer Modul sollte stets alle Daten über die Parameter-Schnittstelle übergeben. Der Zugriff auf globale Variable innerhalb einer Funktion ist nur in Ausnahmefällen akzeptabel, wenn Funktionen eine Vielzahl gemeinsamer Variablen verwenden und damit eine zu große Anzahl von Parametern notwendig wäre. Diese Problematik der prozeduralen Modularisierung wurde durch die objektorientierte Programmierung über Klassen sauber gelöst.

Beispiel einer Funktionsdefinition mit Parameter und Rückgabewert :

```
float quadrat ( float x )
{  return x*x;
}
```

Beispiel für eine nicht modulare Funktion :

```
void quadrat ( )
{  printf( "%f", x*x );
}
```

So ausgeführte Funktionen sind für Bibliotheken ungeeignet.

Funktions-Prototypen :
Funktionsprototypen dienen zur Vorankündigung von Funktionen, ohne konkret den Funktionsblock darzustellen. Die geschlossene Darstellung von Funktionsprototypen am Programmanfang verbessert auch die Lesbarkeit eines Programmtextes.
Die Angabe von Prototypen ist für den Compiler auch immer dann notwendig, wenn die Funktionsdefinition erst nach dem Funktionsaufruf erfolgt, wie bei Bibliotheksfunktionen das der Fall ist.
Diese Trennung von Schnittstelle (Prototyp) und Implementation (Funktionsdefinition) ist für Module und Bibliotheken grundlegend.

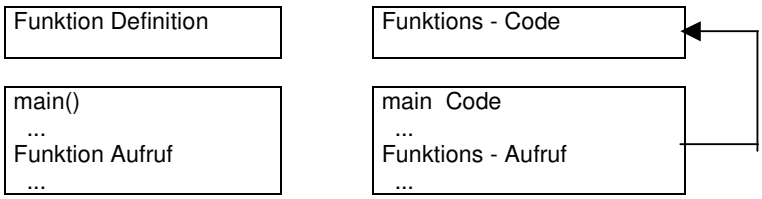
Prototyp Angabe : *ReturnTyp FunktionsName (formale Parameter);*

```
Beispiel :  float quadrat ( float x );
```

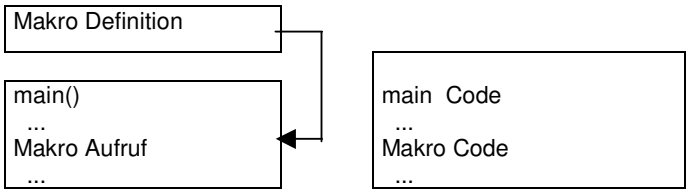
Makros als Alternative zu Funktionen :
Makros sind mitunter die bessere Lösung zur Ausführung von gleichen Programmteilen.
Makros werden als Ersatztext vom Präprozessor an allen Aufrufstellen im Quelltext eingefügt.
Unterprogramme sind Programmteile, die nur einmal im Programmcode vorhanden sind und über einen Unterprogrammsprung aufgerufen werden.
Makros vergrößern einerseits den Programmcode, da jeder Makro-Aufruf als Programmteil hinzugefügt wird, anderseits beschleunigen sie die Ausführung, da kein Unterprogrammsprung notwendig ist.
Makros werden dann vorteilhaft eingesetzt, wenn kurze Programmteile rasch ausgeführt werden sollen.
Makros können jedoch keine lokalen Variablen und Schleifenanweisungen enthalten.

Vergleich der Programm-Ausführung bei Funktionen und Makros :

Funktion : Quellprogramm → Programm Code



Makro : Quellprogramm → Programm Code



Definition von Makros *: #define MakroName (Par1, Par2, ...) Ersatztext*

Die Parameternamen müssen im Ersatztext in runden Klammern stehen.

Beispiel :

```
Definition :  #define  quadrat(x) (x)*(x)

Aufruf :      y= quadrat(2*b);
```

An der Aufrufstelle von **`quadrat(2*b)`** wird der Ersatztext **`(2*b)*(2*b)`** eingesetzt.

Inline-Funktionen (nur bei C++) als Alternative zu Makros :
Inline Funktionen haben ähnliche Vorteile wie Makros. Die Umsetzung von Inline-Funktionen erfolgt jedoch nicht durch den Präprozessor, sondern durch den Compiler. Der Compiler fügt an allen Aufrufstellen den Funktionscode ein. Inline Funktionen sind wie Makros vorteilhaft bei kurzen, rasch auszuführenden Programmteilen.

Definition von Inline-Funktionen :
inline *Returntyp Funktionsname (Parameter)*
{ Anweisungen }

Beispiel :

```
inline float quadrat ( float x )
{  return x*x ;
}
```

Klasse als Modul :

Klassen sind bereits vom Konzept her modular. Klassen enthalten als logische Einheit die Daten und Funktionen für eine bestimmte Aufgabe. Öffentliche Elemente der Klasse stellen dabei die Schnittstelle nach außen dar und private Elemente können innerhalb der Klasse geschützt verwendet werden.

Präprozessor

Präprozessor Anweisungen werden vor dem Compiler ausgeführt und können damit das Programm nur im Quelltext bearbeiten. Über Präprozessor Anweisungen können Definitionen, Einbindungen und Compiler Direktiven ausgeführt werden. Alle Präprozessor Anweisungen beginnen mit dem Zeichen '#' .

Definitionen (defines) : Ersatztexte für Konstante und Makros

#define Bezeichner Ersatztext

Aufhebung der Definition :

#undef Bezeichner

Beispiel :

```
#define NMAX  100
```

Einbindungen (includes) : Dateien im Quelltext einbinden

```
#include <Dateiname>                  // sucht nur im definierten Include-Pfad
#include "Pfadangabe\Dateiname"    // durchsucht in der Pfadangabe und im Include-Pfad
```

Beispiel : Header Datei über include einbinden

```
#include <stdio.h>
```

Beispiel : Datei mit Funktionsdefinitionen über include einbinden

```
#include "statistik.inc"
```

Compiler Direktiven : Anweisungen an den Compiler

```
#if  < Bedingung >
#else
#elif < Bedingung >
#endif
```

Der Compiler compiliert die Zeilen nach der #if - Anweisung nur, wenn die Bedingung gültig ist, andernfalls werden alle Zeilen bis #else oder #endif übersprungen.

```
#ifdef   < Bezeichner >
```

wird mit logisch wahr (=1) ausgewertet, wenn der Bezeichner zuvor definiert wurde

```
#ifndef < Bezeichner >
```

wird mit logisch wahr (=1) ausgewertet, wenn der Bezeichner nicht definiert wurde.

Beispiel : damit Header Dateien nicht mehrfach eingebunden werden, verwendet man folgende Abfragen

```
#ifndef   _STDIO_H
#define  _STDIO_H
    // Prototypen , Definitionen , Makros
#endif
```

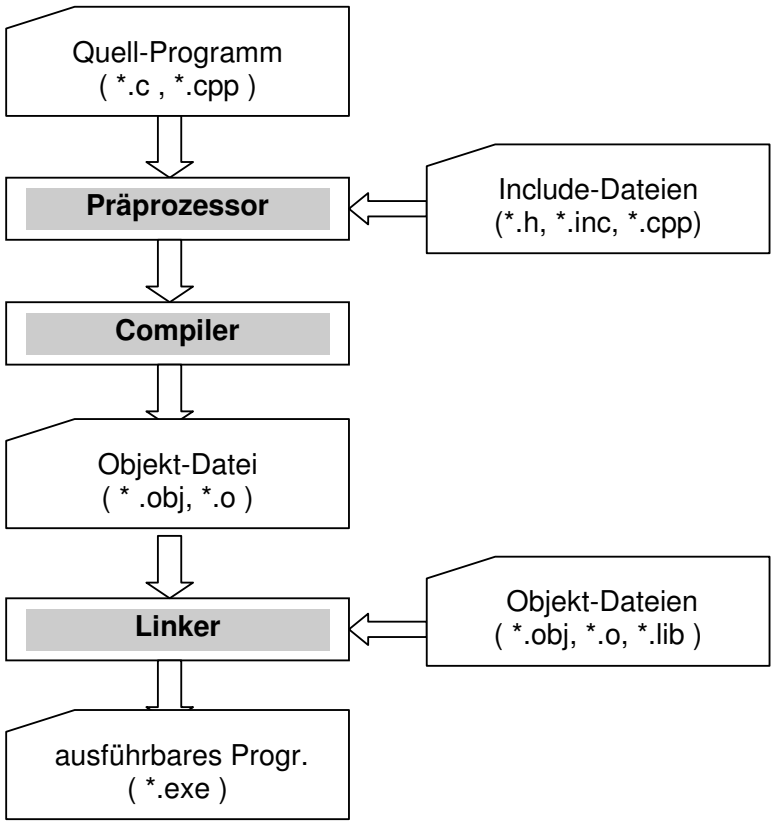
Meldungen und Abbruch der Übersetzung :

```
#error <Meldung>
```

Pragma als implementationsabhängige Direktive :

```
#pragma  <Anweisungsname>
```

Schema der Programmübersetzung bei C/C++ :



Module erstellen und einbinden

Programmteile können entweder im Quelltext über den Präprozessor oder
als übersetzte Objekt-Datei über den Linker in andere Programme eingebunden werden.
Begriffe :

Including : Einbinden von Programmteilen im Quelltext über #include
Linking : Einbinden von Objekt-Dateien über den Linker

Including ist einfacher auszuführen und vorteilhaft in der Entwicklungsphase. Nachteilig ist, dass die eingebundenen Programmteile jedes Mal mit dem Anwendungsprogramm mitkompiliert werden müssen.
Linking ist aufwendiger bei der Erstellung, jedoch vorteilhaft bei der Einbindung, da die bereits übersetzten Objekt-Dateien nur mehr zum Anwendungsprogramm dazugelinkt werden.
Bei großen professionellen Softwareprojekten wird über Linking mit Objekt-Bibliotheken gearbeitet, wie auch bei den C/C++ Standard-Bibliotheken.

Including :

Erstellung von Include-Dateien :

Fertig getestete Funktionsdefinitionen im Quelltext als Datei abspeichern.
(mit Dateinamen *.inc , *.c, *.cpp)

Einbinden von Include-Dateien :

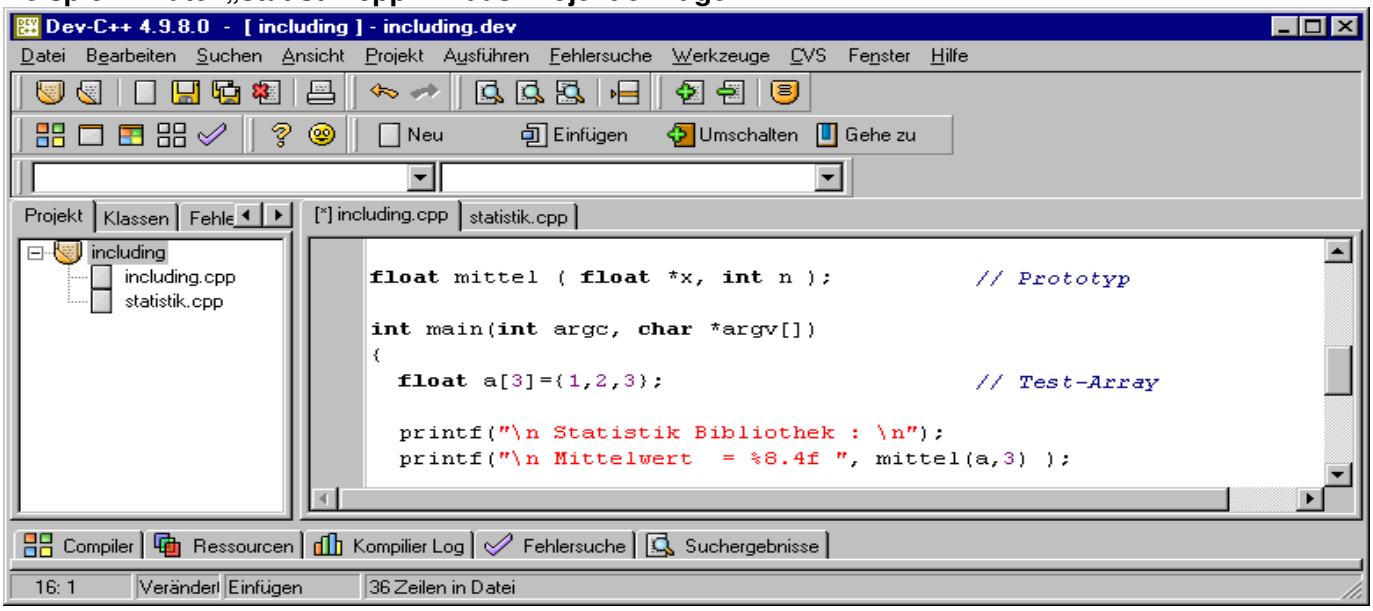
Im Anwendungsprogramm die Include-Dateien über #include-Anweisungen einbinden.

Beispiel : Datei „statistik.cpp“ über Including einbinden

```
/* ------------------------------------------------------------------
   Include Datei    :      statistik.cpp
   ------------------------------------------------------------------
*/
float mittel ( float *x, int n )        // Funktionsdefinition
{ int i;
  float sum=0;
  for (i=0;i<n;i++)
  { sum += x[i];
  }
  return sum/n;
}
  ...

/* ------------------------------------------------------------------
   program name   :       including.cpp
   ------------------------------------------------------------------
*/
#include <stdio.h>

#include "statistik.cpp"                  // Include Datei einbinden

int main(int argc, char *argv[])
{  ...
  printf("\n %8.4f ", mittel(a,3) );     // Funktion aus statistik.cpp verwenden
   ...
}
```

mehrere Programmdateien über ein Projekt einbinden :
Eine Alternative zum Including ist das Einbinden über die Projektverwaltung der IDE (z.B. Dev-C++). Bei einem Projekt können neben dem Hauptprogramm auch weitere Dateien mit Funktionsdefinitionen in das Projekt hinzugefügt werden. Damit im Hauptprogramm die Funktionsaufrufe vom Compiler akzeptiert werden, müssen die Prototypen der Funktionen anderer Dateien angegeben werden.

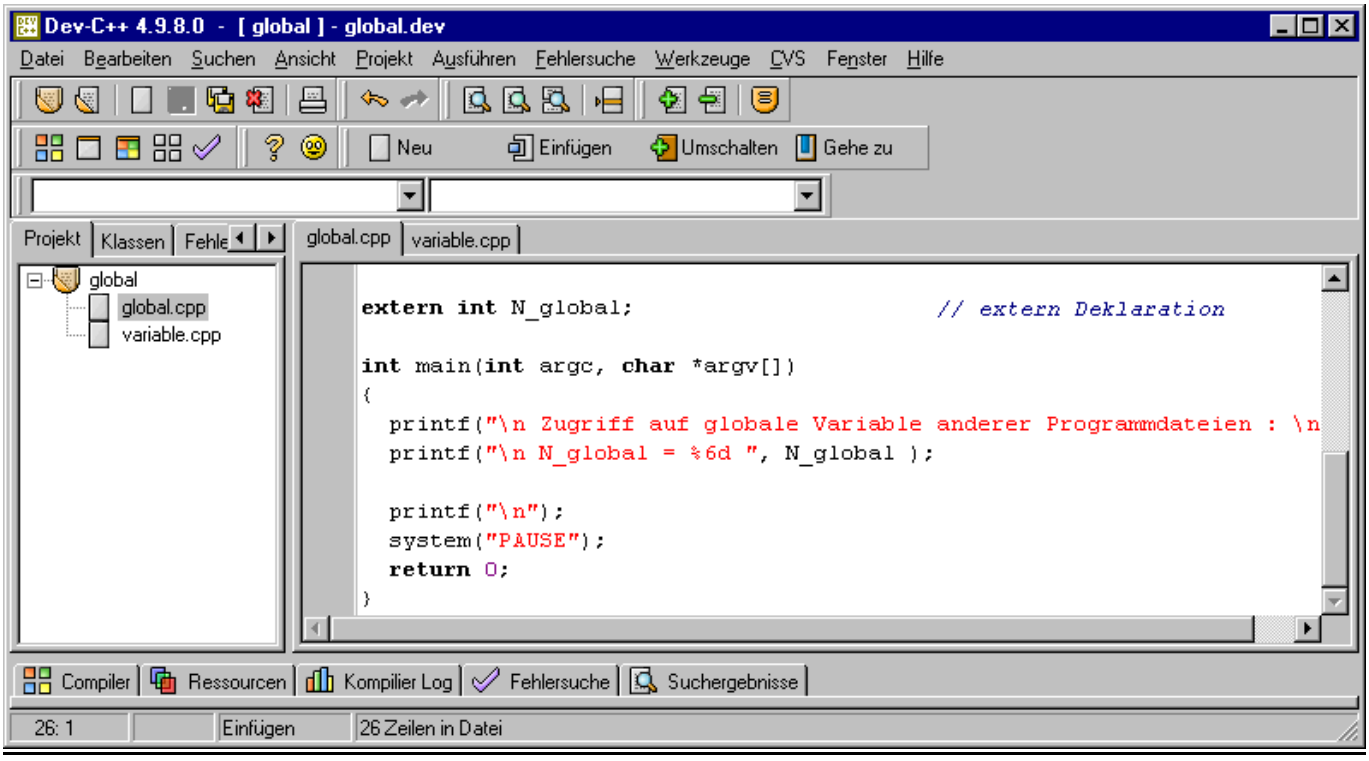
Beispiel : Datei „statistik.cpp“ in das Projekt einfügen



Der Zugriff auf globale Variable einer anderen Programmdatei wird dadurch möglich, indem die Variable über eine extern- Deklaration dem Compiler bekannt gegeben wird.

Beispiel : Zugriff auf globale Variablen anderer Programmdateien

```
/* ------------------------------------------------------------------
   program name  :       variable.cpp
   ------------------------------------------------------------------
*/
int N_global=100;                          // Definition einer globalen Variablen


/* ------------------------------------------------------------------
   program name  :    global.cpp
   ------------------------------------------------------------------
*/
extern int N_global;                          // extern Deklaration

int main(int argc, char *argv[])
{  ...
  printf("\n N_global = %6d ", N_global );// globale Variable verwenden
   ...
}
```



Linking :

Erstellung der Bibliotheks-Objekt-Dateien :

Fertig getestete Funktionsdefinitionen im Quelltext als Datei abspeichern und compilieren.
Der Compiler erzeugt eine Objektdatei mit gleichem Namen und Endung *.obj oder *.o

Erstellung der Bibliotheks-Header-Dateien :

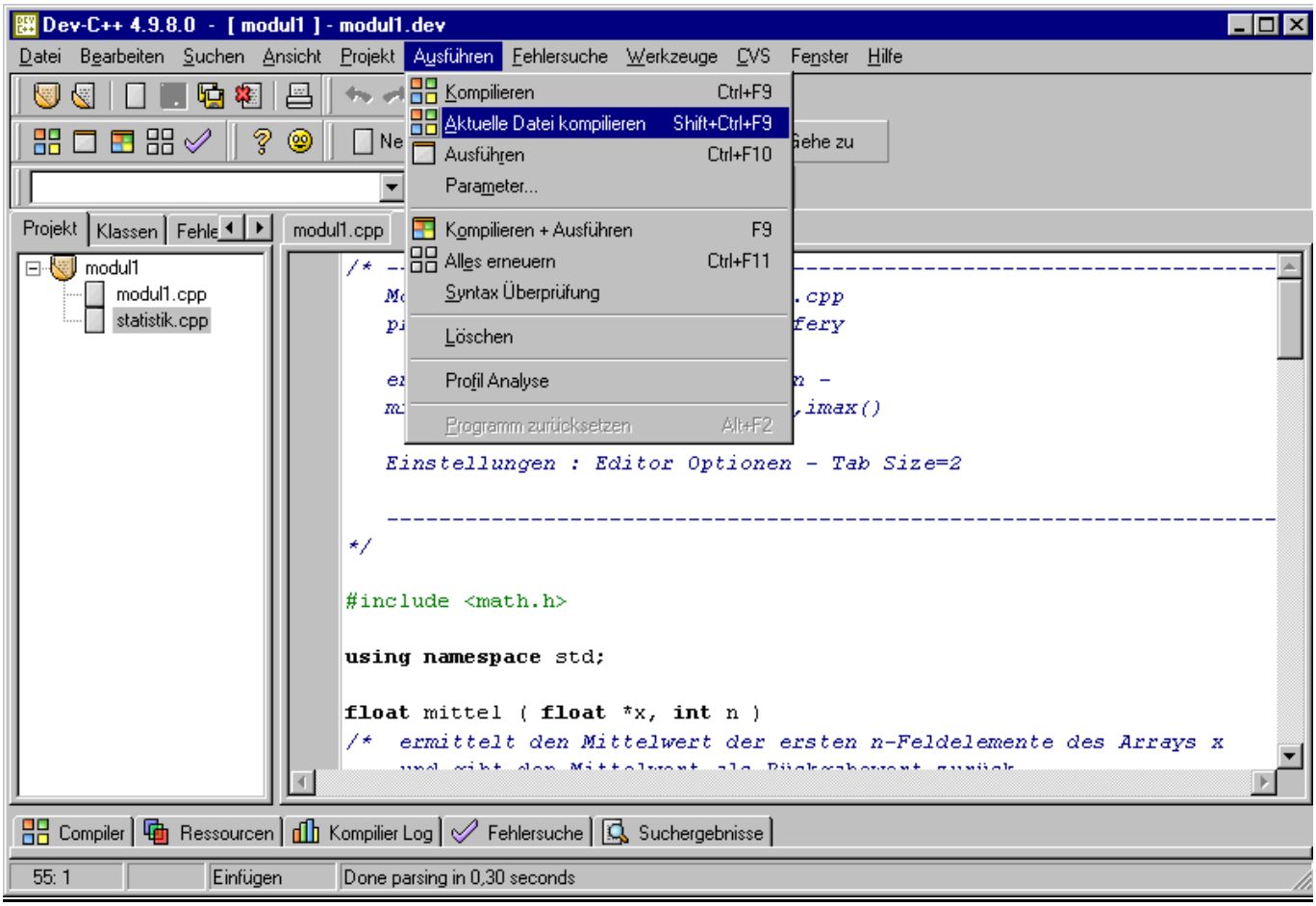
Damit die Verwendung einer Bibliotheks-Objektdatei in anderen Programmen einfach durchführbar wird, ist die Erstellung einer Headerdatei notwendig. Die Headerdatei enthält die Prototypen der Funktionen der Bibliotheks-Objektdatei und eventuell notwendige Defines und Makros.
Headerdateien werden als Quelltextdateien mit der Endung *.h erstellt.

Einbinden von Bibliotheks-Objekt-Dateien :

Im Anwendungsprogramm wird über die #include - Anweisung die Headerdatei der gewünschten Bibliotheks-Datei eingebunden.
Damit zum Anwendungsprogramm die Bibliotheks-Objekt-Datei dazugebunden wird, muss dem Linker die zu bindende Datei bekannt gegeben werden. Das erfolgt am einfachsten über die Projektverwaltung der IDE durch das Hinzufügen der Objekt-Datei in das Projekt.

Beispiel : Objekt-Datei „statistik.o“ erstellen und über Linking einbinden

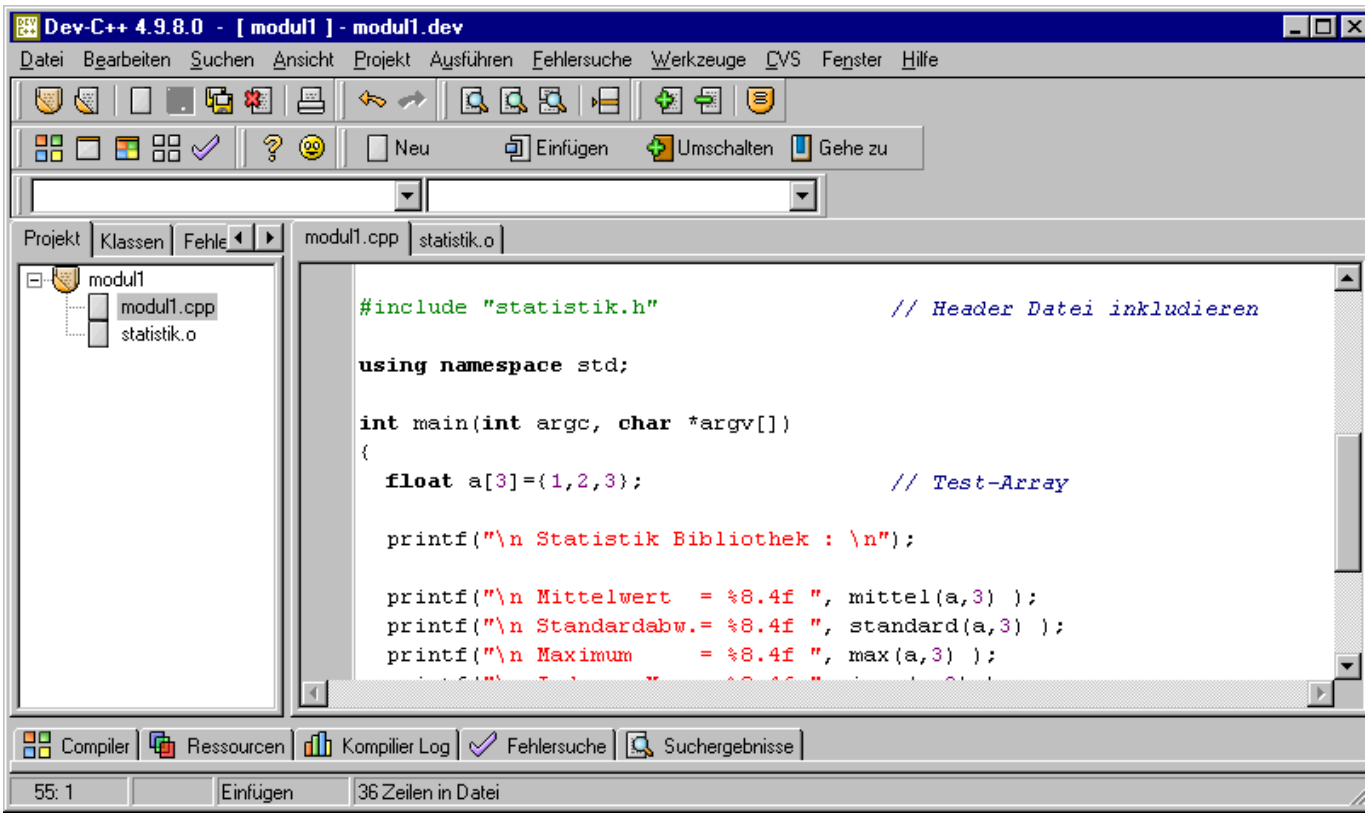
1) Funktionsdefinitionen der Datei „statistik.cpp“ compilieren



2) Erstellung der Headerdatei „statistik.h“

```
/* ------------------------------------------------------------------
   Headerdatei zu Modul Statistik :  statistik.h
   enthält die Prototypen der Statistik-Funktionen -
   ------------------------------------------------------------------
*/

#ifndef statistik_h                    // Abfrage ob bereits eingebunden
#define statistik_h

float mittel ( float *x, int n );      // Funktions-Prototypen
float standard( float x[], int n);
float max( float x[], int n);
float min( float x[], int n);
float imax( float x[], int n);

#endif
```

3) Einbindung der Bibliotheksfunktionen im Anwendungsprogramm „modul1.cpp“ über ein Projekt



Im Programm „modul1.cpp“ wird über die #include - Anweisung die Headerdatei „statistik.h“ eingebunden. Über die Projektverwaltung unter „zum Projekt hinzufügen“ wird dem Anwendungsprogramm die Bibliotheks-Objekt-Datei „statistik.o“ dem Linker bekanntgegeben und beim Übersetzen dazugebunden. Unter Dev-C++ ist es außerdem notwendig unter Projekt-Optionen / Dateien für die Objekt-Datei die Checkbox „ins Linken einbeziehen“ auszuwählen.

Die Projektverwaltung über make :

Das Standard-Dienstprogramm zur C/C++ - Projektverwaltung ist make. (auch unter DevC++)
Über make können alle zu einem Projekt gehörenden Programmdateien compiliert, gelinkt und eine ausführbare Datei (executable File) erstellt werden. Die von make auszuführenden Anweisungen stehen in einem Makefile. Makefiles enthalten alle notwendigen Angaben für das Übersetzen, Aktualisieren und Linken von Programmdateien.

Kommandozeilen-Aufruf : **make** -f <*makefile_name*>

Beispiele :

```
// Pfadeinstellung für Kommandoaufrufe von DevC++ :
C:\>path=c:\programme\dev-cpp\bin
// make - Kurzhilfe ausgeben :
C:\>make  --help
// make - Projektausführung über Makefile
C:\>make -f Makefile
```

Beispiel für ein Makefile :

```
# Bezeichner für Lib-Pfade und Include-Pfade
LIBS =  -L"c:/programme/dev-cpp/lib"
INCS =  -I"c:/programme/dev-cpp/include"

# Produktionsregel für Linker Aufruf
modul1.exe: modul1.o statistik.o
      g++.exe modul1.o statistik.o -o "modul1.exe" $(LIBS)
# Produktionsregel für Compiler Aufruf
modul1.o: modul1.cpp
      g++ -c modul1.cpp -o modul1.o  $(INCS)
```

Beispiel für ein DevC++ - Makefile (Makefile.win) :

```
# Project: modul1
# Makefile created by Dev-C++
CPP  = g++.exe
CC   = gcc.exe
WINDRES = windres.exe
RES  =
OBJ  = modul1.o statistik.o $(RES)
LINKOBJ  = modul1.o statistik.o $(RES)
LIBS =  -L"c:/programme/dev-cpp/lib"
INCS =  -I"c:/programme/dev-cpp/include"
CXXINCS =  -I"c:/programme/dev-cpp/include/c++"  -I"c:/programme/dev-
cpp/include/c++/mingw32"  -I"c:/programme/dev-cpp/include/c++/backward"  -
I"c:/programme/dev-cpp/include"
BIN  = modul1.exe
CXXFLAGS = $(CXXINCS)
CFLAGS = $(INCS)

.PHONY: all all-before all-after clean clean-custom
all: all-before modul1.exe all-after

clean: clean-custom
      rm -f $(OBJ) $(BIN)

$(BIN): $(LINKOBJ)
      $(CPP) $(LINKOBJ) -o "modul1.exe" $(LIBS)

modul1.o: modul1.cpp
      $(CPP) -c modul1.cpp -o modul1.o $(CXXFLAGS)
```

Lib-Files erstellen und verwenden :

Objektdateien sind für große Bibliotheken recht unhandlich, da die Vielzahl von Dateien beim Einbinden unübersichtlich und aufwendig wird. Einzelne Objektdateien mit mehreren Funktionen zu erstellen, ist auch wenig effizient, da Objektdateien nur als Ganzes eingebunden werden und damit nicht verwendete Funktionen mitgeschleppt werden.
Libraries (Lib-Files) enthalten als Archiv mehrere Objektdateien, die logisch zusammengehören.
Der Linker holt aus einem Lib-File nur jene Objektdateien von Funktionen, die im Anwendungsprogramm tatsächlich aufgerufen werden. Lib-Files werden daher so erstellt, dass die einzelnen Objektdateien jeweils nur eine Funktion enthalten.

Die Erstellung von Lib-Files erfolgt über ein Archiv-Tool, wie ar (auch unter DevC++)

Kommandozeilen-Aufruf : **ar** options Archivfile Files, ...

```
Options :
d           - delete file(s) from the archive
r[ab][f][u] - insert new file(s) into the archive or replace existing
t           - display contents of archive
  ...
```

Beispiele :

```
//  Objektdatei max.o ins Lib-File stat.a eintragen
ar -rf  stat.a  max.o
//  Einträge in Lib-File anzeigen
ar -t  stat.a
max.o
//  Objektdatei von Lib-File löschen
ar -d  stat.a  min.o
```

Die Einbindung von Lib-Files erfolgt über eine Linker Option,
die unter DevC++ über den Eintrag unter Projekt Optionen / Parameter / Linker
mit „Bibliothek/Objekt hinzufügen“ und Angabe des Lib-Files ausgeführt wird.