4. 2. Verwaltung großer Projekte

alle Funktionen eines C-Programms in einer Datei:

- bei großen Programmen unübersichtlich
- Wiederverwendung von Programmteilen (Modulen) wird erschwert
- man hat keinen Überblick, was für Funktionen vorkommen
- bei kleinen Änderungen muss alles neu kompiliert werden

Deshalb: Modularisierung

- zusammengehörige Funktionen kommen in eine Datei (Modul)
- mehrere Dateien werden in einem Projekt verwaltet (Projektdatei bei Borland-C, Makefile unter Unix) – Projekt-datei enthält auch Informationen für Compiler und Linker
- die Funktionsköpfe (Header) werden in spezielle Dateien kopiert (und jeweils mit Semikolon abgeschlossen), diese können dann mit **`#include`** in mehrere Module ein-gebunden werden: die wichtigsten Informationen über die Funktionen (Typen der Argumente, des Rückgabewertes) sind dann dort dem Compiler bekannt, so dass diese Funktionen benutzt werden dürfen, ohne dass ihre genaue Definition selbst in der Datei steht
- Header-Dateien (Sammlungen von Funktionsköpfen) bekommen gewöhnlich die Endung **`.h`**, Dateien mit vollständigen Funktionscodes die Endung **`.c`**.

Beispiel für einen Funktionskopf (Header; *Prototyp*):

```
float meine_funktion(float x, int n);
```

vollständige Funktionsdefinition:

```
float meine_funktion(float x, int n)
   {
   float h;
   h = n*x + 1.0;
   return sqrt(h);
   }
```

Die schon bekannten Bibliotheksdateien (**`stdio.h`**, **`string.h`**, **`math.h`** ...) sind solche Header-Dateien (Sammlungen von Funktionsprototypen).

Vereinbarung:
`#include <name.h>` sucht die Datei **`name.h`** in einem Standard-Verzeichnis für Bibliotheksdateien
`#include "name.h"` sucht die Datei **`name.h`** im aktuellen Verzeichnis (sinnvoll bei selbstgeschriebenen Header-Dateien)

Beachte:

> Wenn eine mit **`#include`** eingebundene Datei verändert wird, müssen alle Dateien, in die sie eingebunden ist, neu kompiliert werden.

Fortgeschrittene Entwicklungsumgebungen und die Unix-Makefile-Syntax bieten Tools, um diese Abhängigkeiten automatisch zu überwachen und die erforderlichen Neu-kompilierungen sicherzustellen.

Empfehlung:

> Die Verwendung von Funktions-Prototypen ist auf jeden Fall sinnvoll (auch wenn diese nicht in eine Header-Datei ausgelagert werden), da der Compiler so zu einer besseren Fehlererkennung bei den Funktionsaufrufen befähigt wird.

Header-Dateien werden oft nicht nur für Funktions-Prototypen benutzt, die in mehreren Modulen Verwendung finden, sondern auch für *Konstanten-Definitionen* **(`#define const 1.234`**), *Typdeklarationen* und *Deklarationen globaler Variablen*.

Jedoch: Vorsicht bei *Variablendeklarationen* in Header-Dateien, die im selben Projekt im mehrere Dateien eingebunden werden!

- PROBLEM: Variablen dürfen nur einmal "richtig" deklariert sein
- weitere Deklarationen derselben Variablen (in anderen Programm-Modulen) müssen mit dem vorangestellten Schlüsselwort **`extern`** gekennzeichnet sein.

Abhilfe, wenn globale Variablen in einer gemeinsam von mehreren Modulen genutzten Datei **`allg.h`** deklariert werden sollen:

Deklaration in **`allg.h`**:

```
intext int globale_var;
```

Einbinden von **`allg.h`** in *eines* der Programm-Module (das "Hauptmodul", i.allg. der Teil, der **`main`** enthält):

```
#define intext
#include "allg.h"
#undef intext
```

(der Compiler weist hiermit **`intext`** den leeren String zu)

Einbinden von **`allg.h`** in alle übrigen Programm-Module (wo **`globale_var`** dann als **`extern`** erscheinen muss):

```
#define intext extern
#include "allg.h"
#undef intext
```

`globale_var` darf dann in allen Modulen verwendet werden und hat überall denselben Wert.

(Eine andere Möglichkeit, dasselbe Ziel zu erreichen, besteht in der Verwendung der bedingten Kompilierung mit den Compiler-Direktiven **`#ifdef`** und **`#ifndef`** .)