Modulare Programmierung und Bibliotheken

Alexander Droste

Proseminar: C - Grundlagen und Konzepte, 2013

Inhalt

Definition Modul

Modularität:
- die Aufteilung des Ganzen in Teile
- Gegenteil von monolitisch: das Ganze besteht aus einem

Modul:
- Teil des Ganzen
- weitestgehend unabhängig von anderen Einheiten

Im Kontext der Programmierung:
- funktionale Einheit
- setzt sich aus Implementation & Schnittstelle zusammen (technisch formuliert: Definition & Deklaration)
- Verbindungen zwischen den Modulen sind möglich durch Nutzung ihrer Schnittstellen

Wesen der Modularisierung

Ziele:
- Wiederverwendung
- Abstraktion
- Kombinationsmöglichkeit
- Skalierbarkeit

Vorraussetzungen:
- Kapselung
- Kohäsion

Kapselung

Trennung von Implementation und Schnittstelle

Implemetation:
- enthält die konkreten Ausführungsanweisungen
- wird in .c Dateien geschrieben

Schnittstelle:
- Menge an verwendbaren Funktionen, Variablen, Konstanten die anderen Modulen zur Verfügung stehen
- wird in .h Dateien geschrieben

Implementation und Schnittstelle

```c
//math_functions.c - Implementation
#include <stdio.h>

int add(int first, int second)
{
    return (first + second);
}

int sub(int first, int second)
{
    return (first - second);
}

static void non_interface_function(double d)
{
    printf("%f",d);
}
```

```c
//math_functions.h - Schnittstelle
#include “another_header.h”

int add(int first, int second);
int sub(int first, int second);
```

```c
/*client.c
  Nutzer der Schnittstelle
*/
#include “math_functions.h”

void print_solutions(int i)
{
    printf("%i",i);
}
```

Kapselung Beispiel

```c
//container.c
static container* cont;

void init_and_create_container(char* type, unsigned int size){
    if (strcmp(type,"linked_list") == 0 && cont == NULL){
        cont = create_linked_list_with_size(size);
    }
    else if (strcmp(type,"array") == 0 && cont == NULL){
        cont = create_array_with_size(size);
    }
}

void set_value_at_index(int val, int idx){
    if(strcmp(cont.type, "linked_list") == 0 && cont != NULL){
        set_value_in_linked_list_at(val, idx)
    }
    else if(strcmp(cont.type, "array") == 0 && cont != NULL){
        set_value_in_array_at(val, idx)
    }
}
```

```c
//container.h
void* init_and_create_container(char* type, unsigned int size);
void set_value_at_index(int val, int idx);
```

Kapselung

Modifier welche die Kapselung beeinflussen:

extern:
- Zugriff auf Funktionen und Variablen, die nicht durch Header importiert wurden & sich in anderer Quelldatei befinden
- erneute Deklaration zur Verwendung nötig

```c
//onefile.c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    extern void function_in_second_file();
    extern int var_in_second_file;

    printf("%i\n", var_in_second_file);
    function_in_second_file();
    return 0
}
```

```c
//anotherfile.c
#include <stdio.h>

int var_in_second_file = 21;

void function_in_second_file()
{
    printf("got used\n");
}
```

Kapselung

Modifier welche die Kapselung beeinflussen:

static:
- beschränkt den Sichtbarkeitsbereich auf .c Datei in welcher Funktion/Variable enthalten ist
- Variablen werden permanent im Heap gehalten, Speicher kann aber nicht über “free” wieder freigegeben werden

```c
//onefile.c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    /*auf:
      function_in_second_file() und
      var_in_second_file
      kann nicht mehr über extern
      zugegriffen werden!*/
     return 0
}
```

```c
//anotherfile.c
#include <stdio.h>

static int var_in_second_file = 21;

static void function_in_second_file()
{
    printf("did not get used\n");
}
```

Kapselung

Modifier welche die Kapselung beeinflussen:

static:
- zum Schutz von permanenten Variablen innerhalb einer Datei
- Möglichkeit einmaliges Ausführen von Funktionen zu realisieren

```c
//counter.c
#include <stdio.h>

void one_up()
{
    //wird nur 1x initialisiert,
    //fortlaufend inkrementiert
    int static var = 0;

    if (var == 0) {
        printf("addres %p\n", &var);
    }
    ++var;
}
```

```c
//counter.c

int main(int argc, const char * argv[])
{
    for (int i = 1; i <= 5; ++i)
    {
        one_up();
    }
    return 0
}
```

weitere Möglichkeiten der Mod.

- Structs
- Typedefs

```
typedef struct
{
    int x;
    int y;
} Position;

typedef struct
{
    unsigned int width;
    unsigned int height;
} Size;
```

```
//Zusammensetzung
typedef struct
{
    Position pos;
    Size size;
} Box;
```

- Schachtelung von Funktionen, Makros
- (Objektorientierung, Design Pattern)
- Verwenden und Erstellen von Libraries

Logische Kohäsion Beispiel

GSL - Matrix

```
//gsl_matrix_double.h - Ausschnitt der Schnittstelle

int gsl_matrix_add (gsl_matrix * a, const gsl_matrix * b);
int gsl_matrix_sub (gsl_matrix * a, const gsl_matrix * b);
int gsl_matrix_mul_elements (gsl_matrix * a, const gsl_matrix * b);
int gsl_matrix_div_elements (gsl_matrix * a, const gsl_matrix * b);
int gsl_matrix_scale (gsl_matrix * a, const double x);
int gsl_matrix_add_constant (gsl_matrix * a, const double x);
int gsl_matrix_add_diagonal (gsl_matrix * a, const double x);
```

Negativbeispiel

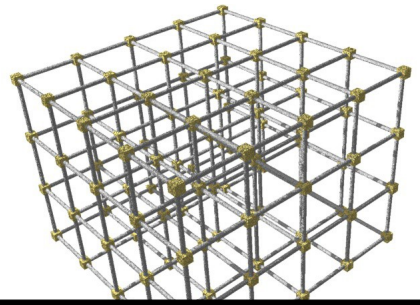
```
//a_bit_of_everything.h

int set_monitor_settings (color_setting* a);
void do_sth (anything* a);
void add_to_database(element * a);
bool export_to_pdf(void* pic_data);
void set_lock (lock* l);
void format_text (void* text);
```

Definition Library

Library/Programmbibliothek:
- stellt eine Schnittstelle von Funktionen für einen bestimmten Problembereich zur Verfügung
  -> die sich für eigenen Code verwenden lässt
- Modul mit hohem Abstraktionsgrad
- im guten Fall mit hoher Kohäsion

Abgrenzung des Frameworks:
- stellt Rahmen und Gerüst für die Programmierung
- enthält mehr Elemente als Libraries



Beispiel für Library

GNU Scientific Library

Complex Numbers, Roots of Polynomials,
Special Functions, Vectors and Matrices,
Permutations, Sorting,
BLAS Support, Linear Algebra,
Eigensystems, Fast Fourier Transforms,
Quadrature, Random Numbers,
Quasi-Random Sequences, Random Distributions,
Statistics, Histograms,
N-Tuples, Monte Carlo Integration,
Simulated Annealing, Differential Equations,
Interpolation, Numerical Differentiation,
Chebyshev Approximation, Series Acceleration,
Discrete Hankel Transforms, Root-Finding,
Minimization, Least-Squares Fitting,
Physical Constants, IEEE Floating-Point,
Discrete Wavelet Transforms, Basis spline

Arten von Libraries

Static:
- fest in das Programm kompiliert
- benötigt viel Ressourcen
- einfache Handhabung
- Endung: .a (Unix), .lib(Windows)

Dynamisch/Shared:
- dyn. Linker lädt & linkt dyn. Libraries
- Speicheradresse zur Library dynamisch
- spart Ressourcen
- können aktualisiert werden
- Endung: .so (Unix), .dylib(Mac), .dll(Windows)

Wo befinden sich Libraries?

Libraries die vom Betriebsystem installiert, verwendet werden:

- /lib

```
a|x /lib ls
apparmor/                libip6tc.so.0            libply-boot-client.so.2
cpp                      libipq.so.0              libply-splash-core.so.2
crda/                    libipq_pic.so.0          libply-splash-graphics.so.2
firmware/                libiptc.so.0             libply.so.2
libcryptsetup.so.4       libnewt.so.0.52          libulockmgr.so.1
libfuse.so.2             libnfsidmap.so.0         libxtables.so.7
libgssglue.so.1          libnl-3.so.200
libip4tc.so.0            libnl-genl-3.so.200
```

Libraries von installierten Programmen oder selbst hinzugefügte :

- /usr/local/lib
- /usr/lib
- /opt

Schnittstellen der Libraries die für eigenen Code verwendet werden können:

- /usr/local/include
- /usr/include

Wie installiere ich zusätzliche Libraries?

Package Manager:
- Linux: apt (Advanced Packaging Tool) -> `sudo apt-get install xy`
- Mac: Homebrew -> `brew install xy`
- Win: win-get

Manuell:
- ./configure; make; sudo make install
- bzw. siehe readme

Vorteile Package Manager:
- automatisierte Installation/Deinstallation
- konsistente Verwaltung
- einfaches Updaten aller Libraries
- Liste von verfügbaren Bibliotheken

Wie kann ich Libraries verwenden?

Implementation sowie Schnittstelle müssen verlinkt werden

- Flags setzen
  Header-Search-Path: -I/pfad/des/include/ordners
  Library-Search-Path: -L/pfad/des/library/ordners
  Name der Library : -l + libraryname

Beispiel: gcc -lgsl gsl_test.c -o gsl_app
gcc -I/desktop/gsl/include/ -L/desktop/gsl/lib/ -lgsl gsl_test.c -o gsl_app

- Flags anzeigen lassen, dynamisch setzen:
  'pkg-config --libs --cflags gsl'

Lazy-Binding
- explizites Öffnen/Schließen durch Anweisungen im Code
- Plugin-Architekturen

Libraryversionen

Programm zwingen eine bestimmte Version, Art zu verwenden:
- vollqualifizierten Pfad verwenden

```
/usr/local/lib/libgsl.so.0.52
/usr/local/lib/libgsl.a
```

Herausfinden welche Version einer dyn. Lib. ein Programm verwendet:

```
otool -L my_app
```

```
my_app:
    /usr/local/opt/gsl/lib/libgsl.0.dylib
    (compatibility version 17.0.0, current version 17.0.0)
    . . .
```

Wie kann ich Libraries erstellen?

- IDEs bieten Presets
- Command-Line (manuell):

statische Libs mit ar (Archiver):

```
gcc -c source_file.c -o object_file.o
ar rcs statLib.a object_file.o
```

dynamische Libs mit gcc:

```
gcc -c -fPIC source_file.c -o object_file.o
gcc -shared -o dynLib.so object_file.o
```

-> Schnittstelle durch Header zur Verfügung stellen

- Build-Tools:
  - GNU build system
  - Cmake
  - SCons

Beispiel mit SCons

- verwendet Build-Scripts in Python
- werden 'SConstruct' genannt
- ausführen von '**scons**' in der Kommandozeile
  -> SCons sucht SConstruct script

```
#SConstruct Script

env = Environment()
env.Replace(CC= 'clang')

lib = File('/usr/local/lib/libgsl.a')
env.Program('my_app', 'hello.c', LIBS=[lib])
#env.StaticLibrary('my_stat_lib', 'hello.c', LIBS='gsl')
env.SharedLibrary('my_dyn_lib', 'hello.c', LIBS='gsl')
```

Lizenzen

GNU General Public License - GPL
- alle Teile der Software müssen bei Veröffentlichung frei sein
- frei: kopieren, verändern, verbreiten.
- kann nicht für proprietäre Software verwendet werden

GNU Lesser General Public License - LGPL
- Teile die frei waren müssen frei bleiben
- kann für proprietäre Software verwendet werden

BSD License - Berkeley Software Distribution License
- keine Verpflichtung Quellcode bei Veröff. anzubieten
- ursprüngliche Entwickler müssen genannt werden
- kann für proprietäre Software verwendet werden

Quellen

Literatur: Titel - Autor - Kapitel

Essentials of Programming Languages - Daniel P. Friedman and Mitchell Wand - [8. Modules]
Practical C Programming - Steve Oualline - [18. Modular Programming]
Computer Systems, A Programmer's Perspective - R. Bryant, D. O'Hallaron - [7. Linking]
An Introduction to GCC - Brian Gough - [3. Compilation options]

Websiten: Thema - Bezeichnung der Seite - Adresse

Modularität - Wikipedia - https://de.wikipedia.org/wiki/Modularit%C3%A4t
Modular Programming in C - embedded -http://tinyurl.com/nrh2yce
Libraries - YoLinux - http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
Shared Libraries - tldp - http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html
Filesystem Hierarchy Standard – Wikipedia - https://de.wikipedia.org/wiki/Filesystem_Hierarchy_Standard
SCons (Build System) - http://www.scons.org/documentation.php
Empirical Comparison of SCons and GNU Make - os.inf.tu-dresden.de/papers_ps/haehne-beleg.pdf
List of software licenses - Wikipedia - https://en.wikipedia.org/wiki/List_of_software_licenses

Bilder
http://programmingtictac.blogspot.de/2013/04/express-module-in-nodejs.html
http://wpteach.com/what-is-a-wordpress-framework-and-why-should-i-care/

Ende der Präsentation

Fragen?