The GAME Engineers

CG #6 – Strukturen, Präprozessoranweisungen und modulare Programmierung

Inhalt

- Präprozessoranweisungen
  - Modulare Programmierung
  - Makros
  - Bedingte Anweisungen

- Strukturen
  - Typdefinitionen
  - Speicherallokation
  - struct

Präprozessoranweisungen – Einführung

```c
#include <stdio.h>

int main(void)
{
    printf("Hallo Welt!");
    return 0;
}
```

Präprozessoranweisungen – Einführung

```c
#include "test.h"

#define APFEL 10
#define SCHOKO int a = foo();
#define MILCH bar(a);

int main(void)
{
    SCHOKO
    MILCH

    return APFEL;
}
```

Präprozessoranweisungen – Einführung

```c
#include "test.c"

#define APFEL 10
#define SCHOKO int a = foo();
#define MILCH bar(a);

int main(void)
{
    SCHOKO
    MILCH

    return APFEL;
}
```

main.c

```c
int foo()
{
    return 1;
}

void bar(char a)
{
}
```

test.c


```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "test.h" 1
int foo()
{
    return 1;
}

void bar(char a)
{
}
# 2 "main.c" 2



int main(void)
{
    int a = foo();
    bar(a);

    return 10;
}
```

Präprozessoranweisungen – Modulare Programmierung

- Modulare Programmierung
  - Aufteilen des Programms in Module
  - In Java wäre das wie das Aufteilen packages (Jigsaw außer Acht gelassen)
  - Sinnvolles Aufteilen in zugehörige Programmabschnitte
  - Ein Modul in C besteht aus eine .c-Datei und einer .h-Datei
    - In .h-Datei stehen Funktionsprototypen, Abhängigkeiten (#include), Konstanten, structs, etc.
    - Die .c-Datei implementiert die eigentliche Logik
  - Einbinden eines Moduls mittels *#include “module.h”*

Präprozessoranweisungen – Modulare Programmierung

```c
#include "marvin.h"

int main(void)
{
    return 0;
}
```
main.c

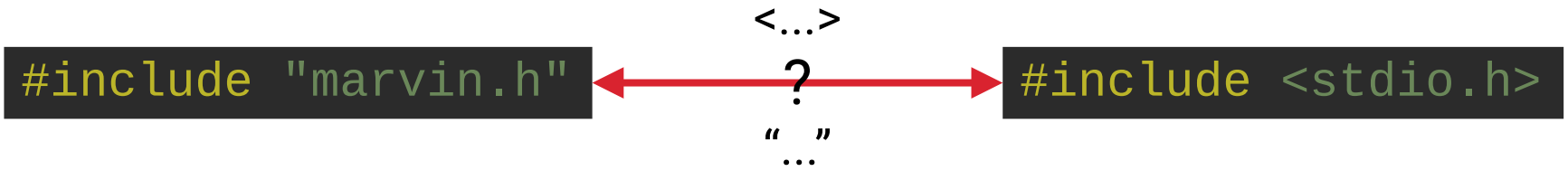
Modul “marvin”

```c
#include <stdio.h>

static int APFEL = 10;
#define BIRNE 20

struct Apfel {
    int a = BIRNE;
    int b = APFEL;
};

int foo(double a);
```
marvin.h

```c
#include "marvin.h"

int foo(double a)
{
    struct Apfel b = { 1, 3 };
    return b.a;
}
```
marvin.c

Achtung:
Da *stdio.h* bereits in *marvin.h* inkludiert wurde, darf es nicht nochmal in *main.c* inkludiert werden!

Präprozessoranweisungen – Modulare Programmierung



- „...“
  - Eigene Module bzw. eigene Dateien, die im gleichen Ordner/Unterordner liegen

- <...>
  - Standarddateien, etc. die irgendwo in PATH zu finden sind
    (Also systemweit zugängliche Dateien)

Präprozessoranweisungen – Makros

- Definierte Ausdrücke, die
  - Konstanten darstellen können
  - Funktionen sein können
  - Etc.

```c
#include <stdio.h>

#define APFEL -1
#define ERROR(err) printf("Fehler: %d", err);

int main(void)
{
    ERROR(APFEL)

    return 0;
}
```

Präprozessoranweisungen – Makros

```
#define ASSERT_VK(f) { \
    VkResult res = (f); \
    if (res != VK_SUCCESS) { \
        printf("Fatal : VkResult is %d in %s at line %d\n",
            res,  __FILE__, __LINE__); \
        assert(res == VK_SUCCESS); \
    } \
}
```

Präprozessoranweisungen – bedingte Anweisungen

```c
#define APFEL

int main(void)
{
    int a;

#ifdef APFEL
    a = 0;
#else
    a = -1;
#endif

    return a;
}
```

```
Process finished
with exit code 0
```

```c

int main(void)
{
    int a;

#ifdef APFEL
    a = 0;
#else
    a = -1;
#endif

    return a;
}
```

```
Process finished
with exit code -1
```

Präprozessoranweisungen – bedingte Anweisungen

- Wenn man *Vulkan* mit *GLFW* nutzen möchte:

```
#define GLFW_INCLUDE_VULKAN
#include "GLFW/glfw3.h"
```



```
#ifdef GLFW_INCLUDE_VULKAN
  #include <vulkan/vulkan.h>
#endif /* Vulkan header */
```

Präprozessoranweisungen – bedingte Anweisungen

```c
#define APFEL 2

int main(void)
{
    int a = -1;

#if APFEL >= 0
    a = 0;
#else
    a = -1;
#endif

    return a;
}
```

```c
#define APFEL -1

int main(void)
{
    int a = -1;

#if APFEL >= 0
    a = 0;
#else
    a = -1;
#endif

    return a;
}
```

Präprozessoranweisungen – bedingte Anweisungen

- Es gibt noch mehr Präprozessoranweisungen, aber das waren die wichtigsten

Strukturen – Typdefinitionen

```c
typedef int apfel;

int main(void)
{
    apfel marvin = 0;

    return 0;
}
```

Strukturen – Typdefinitionen

```
sizeof(expression-or-type)                    size_t
```

```
Also quasi:

size_t sizeof(expr)
{
    // ...
}
```

```
#ifdef _WIN64
__MINGW_EXTENSION typedef unsigned __int64 size_t;
#else
```

```
#define __int64 long long
```


```
Also quasi: typedef unsigned long long size_t;
```

Strukturen – Speicherallokation

```c
#include <stdlib.h>

int main(void)
{
    int *c = malloc(sizeof(int) * 5);

    free(c);

    return 0;
}
```

Strukturen – Speicherallokation

- Anwendung:
  - Übergabe/Zurückgeben eines Arrays
  - Dynamische Arrays
  - Strukturen
  - Quasi Ersatz zu „*new*“ in Java

Strukturen – struct

```c
struct marvin {
    int alter;
    double groesse;
    // ...
};
```

```c
int main(void)
{
    struct marvin m;

    return 0;
}
```

```c
typedef struct marvin {
    int alter;
    double groesse;
    // ...
} marv;

int main(void)
{
    marv m;

    return 0;
}
```

Strukturen – struct

```c
int main(void)
{
    marv m1 = { 5, 0.5 };
    marv m2 = { .alter = 5, .groesse = 0.5 };

    marv m3;
    m3.alter = 5;
    m3.groesse = 0.5;

    return 0;
}
```

Strukturen – struct

```c
int main(void)
{
    marv m1 = { 5 };
    marv m2 = { .groesse = 0.5 };

    return 0;
}
```

```
alter = 5
groesse = uninitialisiert
```

```
alter = uninitialisiert
Groesse = 0.5
```

Strukturen – struct

```c
int main(void)
{
    marv *m = malloc(sizeof(marv));
    m->alter = 5;
    m->groesse = 4;

    return 0;
}
```

Strukturen – struct

```c
typedef struct marvin {
    int alter;
    int groesse;
} marv;

int main(void)
{
    marv m = { 1, 5 };

    int *a = &m.alter;

    printf("alter: %d\n"
           "groesse: %d", a[0], a[1]);

    return 0;
}
```

```
alter: 1
groesse: 5
```

Strukturen – struct

```c
typedef struct marvin {
    int alter;
    double groesse;
} marv;

int main(void)
{
    marv m = { 1, 5 };

    void *a = &m.alter;

    printf("alter: %d\n"
           "groesse: %f", ((int *) a)[0], ((double *) a)[1]);

    return 0;
}
```


```
alter: 1
groesse: 5.000000
```

Strukturen – struct

```c
typedef struct marvin {
    int alter;
    double groesse;
} marv;

int main(void)
{
    void *a = malloc(sizeof(int) + sizeof(double));
    ((int *) a)[0] = 4;
    ((double *) a)[1] = 3.0;

    marv *m = a;

    printf("alter: %d\n"
           "groesse: %f", m->alter, m->groesse);

    return 0;
}
```


```
alter: 4
groesse: 3.000000
```

Strukturen – struct

- Warum so ein Schabernack?
  - In normaler Praxis natürlich nicht unbedingt verbreitet …
  - Aber, wo kann so ein ähnlicher Mechanismus sinnvoll sein?
    - Shader und Buffer (Mapping von Daten)
    - (Auch wenn nicht ganz händisch, sondern automatisch, aber es geht ums Prinzip)

Strukturen – struct

```
float *particles = serializeParticlesystem(ps);
```

```
// ...
glBufferData(GL_SHADER_STORAGE_BUFFER, PARTICLE_AMOUNT * sizeOfParticle,
    particles, GL_DYNAMIC_DRAW);
// ...
```

```
struct particle
{
    float px, py, pz;
    float vx, vy, vz;
    float cx, cy, cz;
    float age;
};

layout(std430, binding = 0) buffer particles
{
    particle p[];
};
```

Ende

Fragen?