FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 6 – Programmstruktur und Namensräume

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Programmstrukturierung
Inhalt

- Programmstrukturierung
  - Was gehört in Header- und Quelldatei?
  - Gültigkeitsbereiche
  - Bibliotheken

Programmstrukturierung

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	consteval	else	mutable	signed	typeid
asm	constexpr	enum	**namespace**	sizeof	typename
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	**using**
case	co_await	false	operator	struct	virtual
catch	co_return	float	private	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Programmstrukturierung

Gültigkeitsbereiche in C

- Welche **Gültigkeitsbereiche** (Scope) kennen Sie in C?
- Aus C bekannt:
  - **Lokal:** Gültigkeit einer Variablen innerhalb eines Blocks. (Block, Funktionsblock, Anweisungsblock)
  - **Global:** Gültigkeit einer Variablen innerhalb einer Quelldatei.

```c
#include <stdio.h>

int a = 1; // globale Variable

int main() {
  // lokale Variable in Funktionsblock
  int b = 2;

  {
    int c = 3; // lokale Variable in Block
    if(true) {
      // lokale Variable in Anweisungsblock
      int d = 4;
      printf("%i\n", a + b + c + d);
    }
  }

  return 0;
} Block
```

Programmstrukturierung
Trennung von Deklaration und Definition

One Definition Rule: Jede Variable, Funktion, Klasse, Konstante usw. hat genau eine Definition.

Quelldatei (.cpp) enthält Definitionen:

- Methoden-/Funktionsdefinition (Implementierung)
- Definition globaler (Member-)Variablen und Konstanten

Headerdatei (.hpp) enthalten möglichst nur Deklarationen:

- Klassendefinitionen
- Methoden-/ Funktionsdeklarationen
- Deklaration von benutzerdefinierten Datentypen wie `enum`, `struct` oder `class`
- Reine Deklaration globaler Variablen und Konstanten mit Hilfe von **`extern`**

Beispiel: `extern int global; extern const int GLOBALE_KONSTANTE;`

- Definition von Konstanten: `const int MAX = 10;`

Tipp: Globale Variablen und Objekte sollten generell vermieden werden, weil Sie für alle zugreifbar sind und Fehler schwer lokalisierbar werden!

Programmstrukturierung
mit Header- und Quelldateien

- In C++ werden Deklaration und Implementierung von Klassen getrennt in **Header-** und **Quelldatei** geschrieben:

Headerdatei: `rechteck.hpp`

```
#ifndef _RECHTECK_HPP_
#define _RECHTECK_HPP_

class Rechteck {
...
};

#endif //_RECHTECK_HPP_
```

Präprozessor Anweisung vermeidet, dass Rechteck doppelt definiert wird.

Quelldatei: `rechteck.cpp`

```
#include "rechteck.hpp"

// Implementierung des Konstruktors
Rechteck::Rechteck(unsigned int b, unsigned int h) {
...
}
// Implementierung der Methode zeichnen()
void Rechteck::zeichnen() {
...
}
```

Headerdatei in der die Definition der Klasse Rechteck gespeichert ist.

Programmstrukturierung
mit Header- und Quelldateien

- Wird eine Klasse von einer anderen Klasse abgeleitet, so wird die Deklaration dieser Klasse durch das Einfügen ihrer Headerdatei zur Verfügung gestellt.

Headerdatei: `quadrat.hpp`

```
#ifndef _QUADRAT_HPP_
#define _QUADRAT_HPP_

#include "rechteck.hpp"

class Quadrat : public Rechteck {
...
};

#endif // _QUADRAT_HPP_
```

Präprozessor Anweisung vermeidet, dass Quadrat doppelt definiert wird.

Quelldatei: `quadrat.cpp`

```
#include "quadrat.hpp"

// Implementierung des Konstruktors
Quadrat::Quadrat(unsigned int groesse){
...
}
// Implementierung der Methode get_groesse()
unsigned int Quadrat::get_groesse() {
...
}
```

Headerdatei in der die Deklaration der Klasse Quadrat gespeichert ist.

Programmstrukturierung
mit Header- und Quelldateien

- Auch wenn eine Instanz einer Klasse erzeugt werden soll, muss ihre Headerdatei eingefügt werden.

Initialisierung mit geschweiften Klammern `{ }` für die Typsicherheit! (ab C++11)

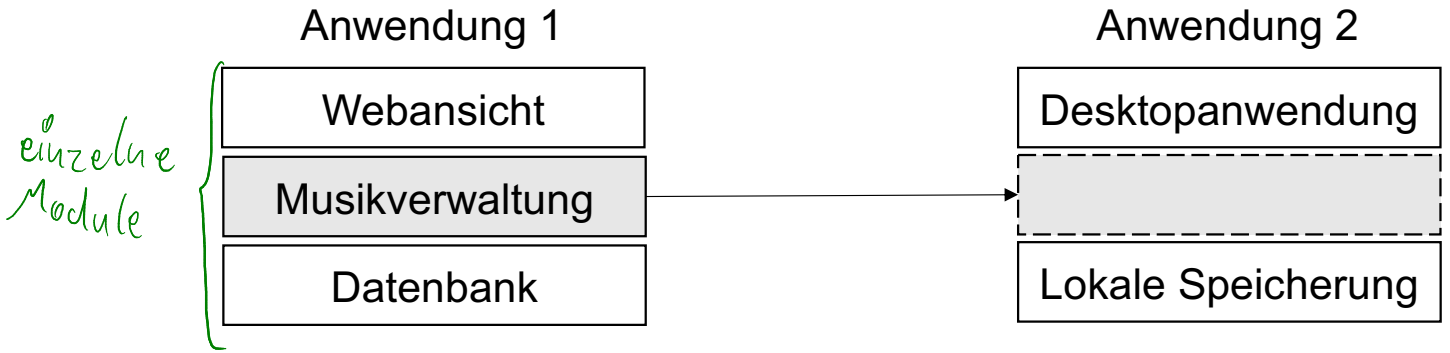
```
#include "quadrat.hpp"
#include "rechteck.hpp"

int main() {
  Quadrat quadrat {2}; // bekannt durch quadrat.hpp
  Rechteck rechteck {2, 3}; // bekannt durch quadrat.hpp oder rechteck.hpp

  return 0;
};
```

Da in `quadrat.hpp` die Headerdatei `rechteck.hpp` eingefügt wurde sind beide Klassen **`Quadrat`** und **`Rechteck`** bekannt.

Programmstrukturierung

Modulare Strukturierung: Rückblick

- **Objektorientierter Entwurf:**
  - Trennung von Aufgaben und Verantwortlichkeiten erfolgt durch
  - ➢ Modulare Struktur



  - ➢ **Vorteil: Wiederverwendbarkeit**

Programmstrukturierung

Modulare Strukturierung in C++

FH MÜNSTER
University of Applied Sciences

- **Klassen** sind die kleinsten Einheiten objektorientierter Programme.
- **Klassen** können auch als **Module** betrachtet werden, da sie alle Anforderungen an ein Modul erfüllen (wenn Sie richtig programmiert sind). Stichwort: Datenkapselung, Geschlossenheit, Schnittstellen etc.

➢ Bei großen Programmen sind Klassen nicht ausreichend, um eine übersichtliche Strukturierung zu erzeugen!
➢ Module können und sollten größer gefasst werden können.

➢ Möglichkeit für *gröbere* Strukturierung in C++?

Programmstrukturierung

Mit Gültigkeitsbereichen

In C++ werden weitere Gültigkeitsbereiche eingeführt:

- **Klasse:** Gültigkeitsbereich vom Anfang der Klassendeklaration bis zum Ende der Klassendeklaration.
- **Namensraum:** Mit einem Namen gekennzeichneter Gültigkeitsbereich (Scope).

Kennzeichnung

- Anfang und Ende der Gültigkeitsbereiche werden wie bei Funktionen mit geschweiften Klammern gekennzeichnet. (Bei Klassen folgt am Ende ein Semikolon)
- Qualifizierer für Gültigkeitsbereiche ist der `::` Operator.

Gültigkeitsbereich

`namespace`-Syntax

- Namensräume markieren die Zusammengehörigkeit von Programmteilen über Klassen- und Funktionsgrenzen hinaus.

```
namespace name{
  // Deklarationen
}
```

```
namespace xyz = name; // Alias eines Namensraum erstellen
```

- Using-Direktiven vereinfachen die Nutzung von Funktionen und Klassen eines Namensraums.

```
using namespace name;
```

```
using name::foo; //Using-Deklaration: lokales Synonym einführen
```

Tipp: Nutzen Sie `using` – Direktiven möglichst nicht in Headerdateien, um Nutzenden der Headerdatei den Namensraum nicht ungewollt zu "vererben".

Gültigkeitsbereiche

Beispiel 1: `namespace`

Wie lautet hier die Ausgabe?

```
#include <iostream>

namespace nsp_a
{
  char foo(){ return 'a'; }
}

namespace nsp_b
{
  char foo() { return 'b'; }
}

int main()
{
  std::cout << nsp_a::foo() << std::endl;
  using namespace nsp_a;
  std::cout << foo() << std::endl;
  std::cout << nsp_b::foo() << std::endl;

  return 0;
}
```

Gültigkeitsbereiche

Beispiel 2: `namespace`

Wie lautet hier die Ausgabe?

```
#include <iostream>

namespace nsp
{
  char foo1(){ return '1'; }
}

namespace nsp
{
  char foo2(1) { return '2'; }
}

int main()
{
  std::cout << nsp::foo1() << std::endl;
  std::cout << nsp::foo2() << std::endl;
  return 0;
}
```

Gültigkeitsbereiche

Beispiel 3: `class`

Wie lautet hier die Ausgabe?

```
#include <iostream>

namespace nsp
{
  class MeineKlasse
  {
    public:
      class MeineInnereKlasse
      {
        public:
          static char foo() { return 'a'; }
      };
      static char foo() { return 'b'; }
  };
} // namespace nsp

using namespace std;
int main()
{
  cout << nsp::MeineKlasse::foo() << endl;
  cout << nsp::MeineKlasse::MeineInnereKlasse::foo() << endl;
  return 0;
}
```

Programmstrukturierung
Bibliotheken

- Werden Programmteile von verschiedenen Teams bearbeitet, genügt manchmal auch die Strukturierung durch Module nicht, um Programmteile von einander abzugrenzen.

  ➢ Lösung: Auslagerung in Bibliotheken

- Zwei Arten:
  - Statische Bibliothek: Sammlung von Objektdateien der Programmteils. Einbindung von Funktionen zur Kompilierzeit.
  - Dynamische Bibliothek: Fertig gebauter Programmteil (ähnlich einer ausführbaren Datei). Einbindung von Funktionen zur Laufzeit.

C++ Bibliotheken

Erzeugen mit GCC

Operating System	Dynamic library	Static library	Library prefix
FreeBSD	.so	.a	lib
macOS	.dylib	.a	n/a
Linux	.so	.a	lib
Windows	.dll	.lib	n/a

https://wiki.freepascal.org/macOS_Static_Libraries, 6.1.2021

Beispiel

1. Objektdatei/en erzeugen

```
g++ -c foo.cpp -o foo.o
```

2a. Statische Bibliothek

```
ar rcs libfoo.a foo.o # Erzeuge Archiv aus allen Objektdateien
```

2b. Dynamische Bibliothek

```
g++ -shared -o libfoo.so foo.o # Erzeugung mit shared-Flag
```

Unix/Linux/MinGW

C++ Bibliotheken

Einbinden mit GCC

- Statisch/Dynamisch:

Beispiel

```
g++ main.cpp -L Pfad/zur/Bib/ -lfoo
```

Unix/Linux/MinGW

- **Anmerkungen zu Dynamischen Bibliotheken:**
  - Ist ein Programm mit einer Dynamischen Bibliothek verlinkt, enthält dieses Programm nicht die Implementierung der Bibliotheksfunktionen.
  - **Vorteil:** das Programm ist schlanker.
  - **Nachteil:** das Programm ist ohne die Bibliothek nicht lauffähig.

Exkurs: Kompatibilität zu C

Einbinden von C-Funktionen/Header

- Falls C-Funktionen in C++ Quelltext eingebunden werden sollen. Der Quelltext innerhalb von `extern "C" {}` wird als C kompiliert.

```
extern "C" {
// C-Funktionsdeklaration
// auch #include ...
}
```

- Falls C-Funktionen sowohl in C als auch in C++ genutzt werden sollen, kann die Einbindung von `extern "C"` über das Makro `__cplusplus` gesteuert werden.

```
#ifdef __cplusplus
extern "C" {
#endif
// C-Funktionsdeklaration
// auch #include ...
#ifdef __cplusplus
}
#endif
```

FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de