FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 3 – Grundlegende Syntaxelemente von C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Grundlagen
Inhalt

- Grundlegende C++ Syntax
  - Native Datentypen
  - Variablen und Konstanten
  - Ausdrücke
- Weitere benutzerdefinierte Datentypen (enum class)
- Typumwandlungen
- Felder und Zeichenketten
- Kontrollstrukturen (insb. bereichsbasierte for-Schleifen)
- Zeiger und Referenzen
- Funktionen

Grundlagen
Datentypen und Konstanten

Datentypen und Konstanten

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	

Datentypen und Konstanten

Ganzzahldatentypen

		Bezeichung	Alternative Bezeichnung	Bit	Literal	
Normale Ganzzahl	**signed** oder **unsigned**	**short**	short int	*16*	**short**(9)	
		int	int	*32*	*9*	
		long	long int	*64*	*9L*	
		long long	long long int	*mindestens 64*	*9LL*	
Zeichen*		**char**		*8*	*'a'*	*siehe ASCII Tabelle*
Logisch**		**bool**		*8*	**true**, **false**	

*Der Zeichendatentyp **char** lässt sich auch als 8 bit Ganzzahldatentyp interpretieren.
** Der logische Datentyp **bool** ist ein besonderer Ganzzahldatentype bei dem 0 als **false** und alles andere als **true** gewertet wird.

Nähere Informationen zu Datentypen auf dem aktuellen System sind über die Headerdatei <limits> zu finden.

Datentypen und Konstanten

Gleitkommadatentypen

FH MÜNSTER
University of Applied Sciences

	Bezeichung	Alternative Bezeichnung	Bit	Literal	Genauigkeit*
Gleitkommazahl	`float`		*32*	*1.f*	*ca. 7*
	`double`		*64*	*1.*	*ca. 16*
	`long double`		*80*	*1.L*	*ca. 19*

*Genauigkeit bedeutet auf wie viele Vor- und Nachkommastellen genau ein Wert angegeben werden kann.

Nähere Informationen zur Datentypen auf dem aktuellen System sind über die Headerdatei `<limits>` zu finden.

Datentypen und Konstanten

Konstanten

- Wird einem Typ das Schlüsselwort **`const`** vorangestellt, handelt es sich um eine Konstante.
- Konstanten sind Zahlen oder Datenstrukturen, die nicht verändert werden können.
- Werden sie einmal mit **`const`** definiert, können sie an mehreren Stellen verwendet werden.
- Konstanten werden meistens groß geschrieben, um sie von Variablen zu unterscheiden.
- Auch Literale sind Konstanten. Üblicherweise werden Literale zur Initialisierung von Variablen genutzt. Werden Literale im Code als Konstanten genutzt, kann es zu Fehlern kommen. Diese sogenannten "Magic Numbers" sind zu vermeiden!

Beispiel:

```
// Definition einer Konstanten
const double PI {3.1415926}; // besser noch ist M_PI aus der Standardbibliothek zu nutzen

kreisflaeche = 3.1415926*r*r; // "Magic Numbers" sollten vermieden werden
kreisflaeche = PI*r*r; // besser ist die Verwendung von definierten Konstanten
```

Datentypen und Konstanten

Das Schlüsselwort `void`

- Das Schlüsselwort `void` ist reserviert für den Fall wenn kein Datentyp bekannt ist oder kein Datentyp zurückgegeben werden soll.
- `void` ist daher kein Datentyp, sondern eher ein Platzhalter.
- Da `void` einen unbekannten Datentyp deklariert, kann für `void` auch kein Speicher reserviert werden.
- `void` ist nützlich bei der Deklaration von Funktionen und Zeigern

  ➢ Dazu später mehr

Grundlagen
Ausdrücke

Ausdrücke

Bilden von Ausdrücken

- Ein Ausdruck besteht aus mehreren Operanden, die durch Operatoren miteinander verknüpft sind.
- Üblicherweise werden Ausdrücke in C++ genutzt um mathematische Ausdrücke im Programm abzubilden.

Beispiel:

```
kreisflaeche = PI*r*r; // Dies ist ein Ausdruck
```

Ausdrücke
Rangfolge von Operatoren

- Im Allgemeinen gelten Vorrangregeln der Algebra, d.h. Punkt vor Strichrechnung inklusive Klammerregeln.

Beispiel:

```
cQuadrat = a * a + b * b; // Satz des Pythagoras
irgendeinErgebnis = a * ( a + b ) * b; // Klammern haben Vorrang
```

- Da es weit mehr Operatoren gibt also Punkt, Strich und Klammern, ist eine genaue Priorisierung der Operationen notwendig.
- Eine vollständige Aufstellung der Rangfolge von Operatoren kann folgender Tabelle entnommen werden:
  https://en.cppreference.com/w/cpp/language/operator_precedence

Grundlagen
Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen

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	

Benutzerdefinierte Datentypen

Strukturen (`struct`)

Eigenschaften von C++ Strukturen	**C++**	**C**

Direkte Initialisierung	`struct Foo{`
	`  int a = 1;`
	`};`
		`struct Foo{`
		`  int a;`
		`};`

Member-Funktionen	`struct Foo{`
	`  int a = 1;`
	`  int b = 2;`
	`  void print(){`
	`  cout << a << " " << b << endl;`
	`  }`
	`};`	

Deklarierung ohne `struct` Schlüsselwort möglich.	`Foo foo;`	`struct Foo foo;`

Benutzerdefinierte Datentypen

Aufzählungstyp (`enum class`)

- `enum class` oder auch äquivalent `enum struct` gibt es seit C++11

Beispiele:

```
//Deklaration
enum class Farbtyp {rot, gruen, blau};

//Variablendefinition und Initialisierung
Farbtyp farbe {Farbtyp::gruen};

//Deklaration mit Abweichung von Standardaufzaehlung 0, 1, 2
enum class Farbtyp {rot=-10, gruen=4, blau=2};

//Deklaration mit Variablendefinition (anonyme Typdefinition)
enum class Farbtyp {rot, gruen, blau} farbe;

//Deklaration mit Abweichung vom Standartyp int (Datentyp muss ganzzahlig sein
enum class Farbtyp : unsigned int {rot, gruen, blau};
```

Benutzerdefinierte Datentypen

Aufzählungstyp: Beispiel

Ist dieser Code fehlerfrei?

```
// beispiel_enum.cpp

int main()
{
    enum class Farbtyp {rot, gruen, blau};
    int i = rot + gruen;
    return 0;
}
```

Benutzerdefinierte Datentypen
sonstige

- Unions (**`union`**): Sehr speziell, siehe Literatur

Beispiel für interessierte:

```
#include <iostream>

union Ascii //Deklaration (Hinweis: union kann auch anonym deklariert werden)
{
  short code {}; // 2 Byte
  char letter; //1 Byte
};

int main()
{
  Ascii ascii; // Für ascii wird nur der Speicher des größten Datentyps angelegt, hier: 2 Byte
  ascii.code = 0101; // initialisiere mit Oktalcode 101
  // verändert code und letter, da beide auf denselben Speicher zugreifen
  std::cout << ascii.letter << std::endl; // Ausgabe: A
  return 0;
}
```

➢ Klasse (**`class`**): Siehe Objektorientierung 1 und 2!

Grundlagen
Zeiger und Referenzen

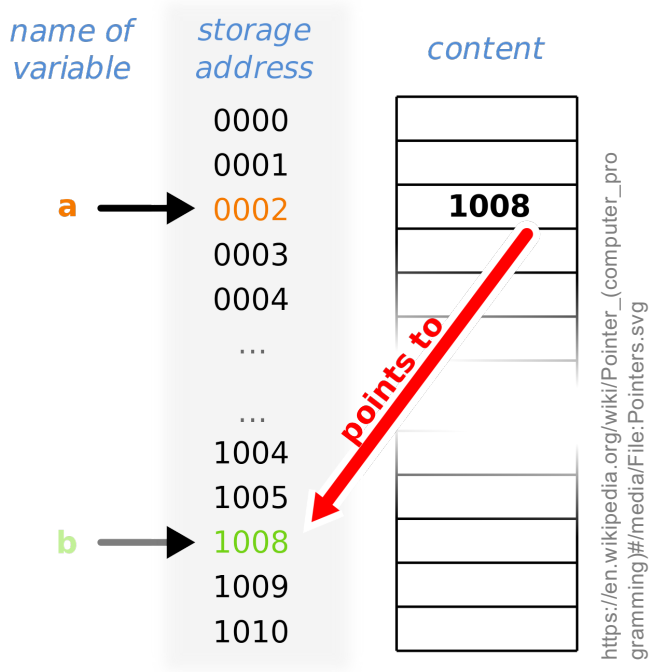
Zeiger

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	

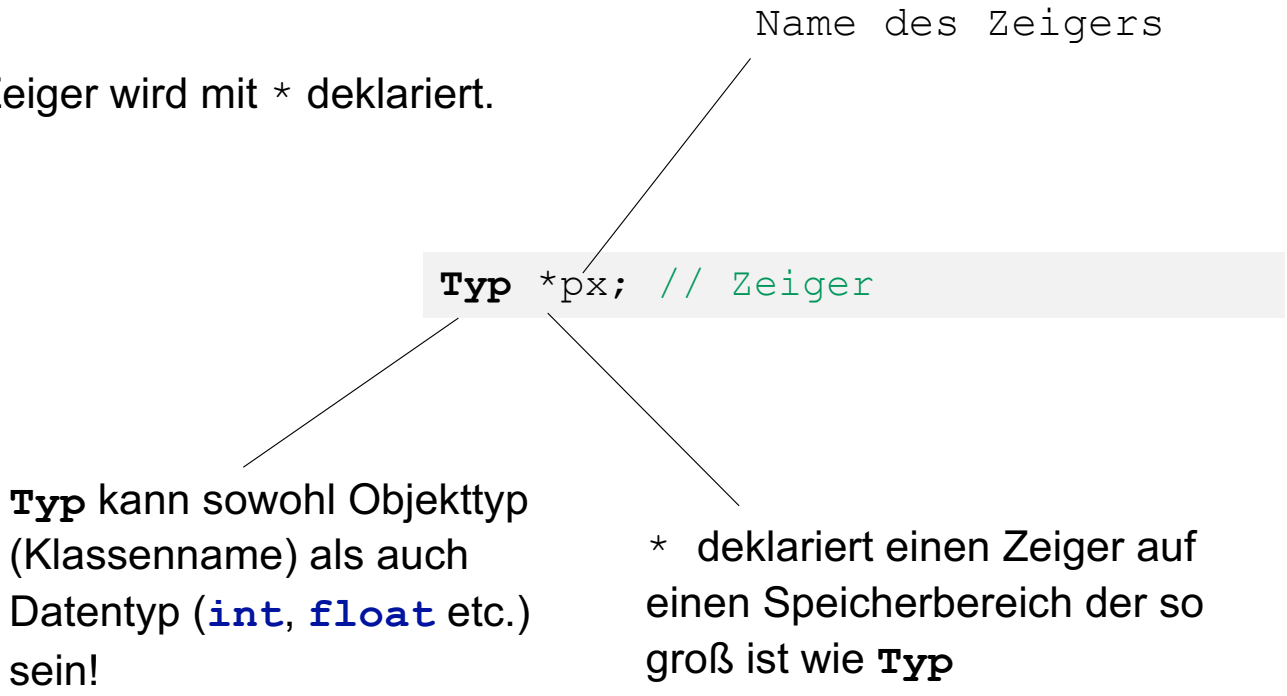
Zeiger
Was ist das? (Wiederholung)

- Zeiger sind spezielle Variablen.
- Zeiger haben wie Variablen
  - einen Namen
  - einen Wert
  - können mit Operatoren verändert werden
- Anders als bei Variablen wird der Wert eines Zeigers
  - als Adresse behandelt



Zeiger
Deklaration

- Ein Zeiger wird mit `*` deklariert.



Zeiger
Adressoperator

- Der Adressoperator **`&`** erzeugt einen Zeiger auf den Speicher einer Variablen.

```
Typ *px; // Zeiger
Typ x; // Variable
px = &x; // Zeiger auf x
```

Adressoperator erzeugt einen Zeiger auf die Variable `x`

Zeiger
Dereferenzierung

- Den Zugriff auf den Wert hinter einem Zeiger nennt man Dereferenzierung

```
Typ *px; // Zeiger
Typ x; // Variable
px = &x; // Zeiger auf x
*px; // Dereferenzierung
```

`*` dient zu Dereferenzierung von px

Zeiger

Das Schlüsselwort **`nullptr`**

- Ein Zeiger wird mit dem Schlüsselwort **`nullptr`** initialisiert, wenn noch keine Speicheradresse zugewiesen wurde.

```
Typ *pA = nullptr;
```

- *Hinweis: Wird ein Zeiger mit `0` oder `NULL` initialisiert, ist in manchen Kontexten durch implizite Typumwandlung nicht mehr ersichtlich, dass es sich um einen Zeiger handelt, da 0 auch als Ganzzahlwert interpretiert werden kann.*

Tipp: **`NULL`** ist ein Macro und kann abhängig von der Implementierung des C++ Standards `0` bedeuten oder **`nullptr`**, daher sollte möglichst **`nullptr`** statt **`NULL`** benutzt werden.

Zeiger und Referenzen

Beispiel: C vs C++

- In **C** erfolgt der Zugriff auf Daten im Speicher über:
  - den Variablennamen
  - einen Zeiger

```
int x = 1; // Variable
int *px = &x; // Zeiger

printf("%i\n", x);
printf("%i\n", *px);

// Ausgabe:
// 1
// 1
```

- In **C++** gibt es drei Möglichkeiten des Zugriffs:
  - den Variablennamen
  - einen Zeiger
  - die Referenz

```
int x {2}; // Variable
int *px {&x}; // Zeiger
int &_x {x}; // Referenz

cout << x << endl;
cout << *px << endl;
cout << _x << endl;

// Ausgabe:
// 2
// 2
// 2
```

Referenz
Deklaration und Initialisierung

- Eine Referenz wird mit `&` deklariert.

```
Typ x;
Typ &rX {x}; // Referenz
```

`&` deklariert eine Referenz auf die Variable x

- Merke: Referenzen müssen immer mit einer Variablen initialisiert werden!

Referenzen und Zeiger als Parameter

Call-By-Value und Call-By-Reference

Call-By-Value

```
void foo(int x) {
  x = 1;
}
```

Funktionsaufruf per Wert:
Eine Veränderung von `x` wirkt sich nur innerhalb von `foo()` aus.

Call-By-Reference (mit Zeiger)

```
void foo(int* px) {
  *px = 1;
}
```

Funktionsaufruf per Wert (aber als Zeiger):
Per Dereferenzierung kann der Wert auf den `px` zeigt auch außerhalb von `foo()` verändert werden.

Call-By-Reference (mit Referenz)

```
void foo(int& rx) {
  rx = 1;
}
```

Funktionsaufruf per Referenz:
Eine Veränderung von `rx` wirkt sich auch außerhalb von `foo()` aus.

Grundlagen
Typumwandlung

Typumwandlung

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	

Typumwandlung
Implizite Typumwandlung

In C/C++ gibt es Standard-Typumwandlungen (englisch: ***type cast*** oder einfach ***cast***),
die "ohne dass man es merkt" ***implizit*** durchgeführt werden.

Beispiel:

```
int a, b;
a = 4.0; // keine Fehlermeldung und keine Warnung, a wird der Wert 4 zugewiesen
b = 4.3; // keine Fehlermeldung aber eine Warnung, b wird der Wert 4 zugewiesen
```

Dies kann zu Informationsverlusten führen:

- Verlust der Genauigkeit, z.B. wenn eine **`double`** Zahl in **`float`** umgewandelt wird.
- Ungewollte Überschreitung von Zahlenbereichen, z.B. Umwandlung von **`int`** in **`short int`**.
- Ebenso können auch Grenzbereiche von Fließkommazahlen ungewollt überschritten werden, z.B. bei Umwandlung von **`double`** in **`float`**.
- Verlust der Nachkommastellen, z.B. wenn **`double`** in **`int`** umgewandelt wird.
- Vorzeichenverlust und Wertänderung, z.B. wenn eine negative **`int`** Zahl in **`unsigned int`** umgewandelt wird.

➢ **Handlungsempfehlung:** Compiler-Warnungen beachten und beseitigen.
Typumwandlungen immer bewusst, d.h. ***explizit*** durchführen

Typumwandlung
Explizite Typumwandlung

- Um Fehler zu vermeiden, sollten ***explizite*** Typumwandlungen (type casts) genutzt werden!
- Die zwei geläufigsten Notationen sind:

```
(Neuer_Typ) Ausdruck
```

```
Neuer_Typ (Ausdruck)
```

Beispiel:

```
int a, b;
long long c;
a = (int) 4.3; // C-Stil Typumwandlung
b = int (4.3); // Typumwandlung mit funktionaler Notation
c = long long (4.3); // Fehler: Klammern nicht vergessen bei zusammengesetzten Typen!
c = (long long) (4.3); // Richtig!
```

- Beide Notationen sind equivalent zueinander.

➢ (-) Nachteil: Diese Typumwandlung ist nicht sicher, da sie nicht zwischen verschiedenen Typumwandlungen unterscheidet und sollte in C++ nur in **Ausnahmefällen** genutzt werden.

Typumwandlung

Explizite Typumwandlung mit `..._cast`

- C++ bietet explizite Typumwandlungen: **`static_cast`**, **`dynamic_cast`** , **`const_cast`** und **`reinterpret_cast`**.
- (+) Vorteil: **Spezifischer:** Die Art der Typumwandlung kann ausgewählt und damit kontrolliert werden.
  Besser erkennbar: Die neuen C++ Typumwandlungen sind im Code besser erkennbar als die "alte" C-Stil Notation.

```
static_cast<Neuer_Typ>(Ausdruck)
```

Ist dazu gedacht kontrolliert Typumwandlungen durchzuführen oder rückgängig zu machen. Kontrolle läuft zur Kompilierzeit.

```
dynamic_cast<Neuer_Typ>(Ausdruck)
```

Typ-Kontrollen werden zur Laufzeit durchgeführt ("dynamic"). Wird im Kontext von Vererbung und Polymorphie verwendet.

➢ **hierzu später mehr**

Vorsicht

```
const_cast<Neuer_Typ>(Ausdruck)
```

Kann die **`const`** Eigenschaft beseitigen.

```
reinterpret_cast<Neuer_Typ>(Ausdruck)
```

Erlaubt jede Typumwandlung ohne Kontrollen außer **`const`**.

Grundlagen
Felder und Zeichenketten

Felder und Zeichenketten

Deklaration und Definition

- Felder dienen dazu eine bestimmte Anzahl an Werten gleichen Typs zu speichern

Beispiel:

```
int wert[10];
```

- Zeichenketten sind Felder die aus Zeichen i.d.R. vom Typ `char` bestehen

Beispiel:

```
char zeichen[10];
```

Felder und Zeichenketten

Initialisierung

- Auch Felder können initialisiert werden (Initialisierung = Definition und Zuweisung)

Beispiele:

```
double wertA[5] {1.0, 2.4, 0.1, -0.2, 1.3}; // Feld initialisieren
double wertB[] {1.0, 2.4, 0.1, -0.2, 1.3}; // Feldgröße wird automatisch erkannt

int mehrdim[][3][2] {
                    {{1}, {3,4}, {5,6}},
                    {{7,8}, {9,10}, {11,12}},
};

char zeichenA[] {'a', 'b', 'c'}; // Zeichenkette initialisieren
char zeichenB[] {"abc"}; // Zeichenkette mit C-String initialisieren
//Achtung: C-String hat immer ein '\0' Literal angehängt
char zeichenC[] {'a', 'b', 'c', '\0'}; // ziechenC äquivalent zu zeichenB
```

Felder und Zeichenketten

Indizierung

- **Achtung:** Genau wie in C überprüft C++ keine Feldgrenzen! Bei Überschreitung können ungewollt Speicherbereiche überschrieben werden.

Beispiel:

```
int wert[10];
cout << wert[15] << endl;
// kein Fehler, da nicht überprüft wird, ob Speicherbereich überschritten wird!
```

Felder und Zeichenketten

C++ Daten-Container (`std::vector`)

- **`std::vector<Typ>`** ist ein sogenannter Daten-Container aus der C++ Standardbibliothek
- Die Einbindung erfolgt über **`#include<vector>`**
- Ein Vektor (`std::vector`) kann genauso genutzt werden wie ein Daten-Feld, schützt aber vor Speicherüberschreitungen und verfügt über viele nützliche Funktionen.

Beispiele:

```
std::vector<double> vec1; // Definiert einen (leeren) Vektor für double Elemente
std::vector<int> vec2 {1, -2}; // Ein Vektor-Container lässt sich wie ein Feld initialisieren
std::cout << vec2[1] << std::endl; // Zugriff auf das zweite Element in vec2
// Ausgabe: -2
std::cout << vec2.size() << std::endl; // Gibt die Länge des Vektors zurück
// Ausgabe: 2
vec2.push_back(10); // fügt ein weiteres Element mit dem Wert 10 hinzu
```

Felder und Zeichenketten

Zeichenketten im C++-Stil (`std::string`)

- In C++ werden Zeichenketten als **`std::string`** gespeichert und verarbeitet.
- Einbindung erfolgt über **`#include<string>`**

Beispiel:

```
std::string str{"Hallo, Welt!"}; // Initialisierung einer String Zeichenkette
std::cout << str << std::endl;
// Ausgabe: Hallo, Welt
```

➢ **Handlungsempfehlung**: In C++ sollten immer Daten-Container und Strings aus der Standardbibliothek genutzt werden, statt einfache Felder und Zeichenketten im C-Stil. Dies erhöht die Sicherheit, Verständlichkeit und Wartbarkeit des Quelltextes!

Grundlagen
Kontrollstrukturen

Kontrollstrukturen

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	

Kontrollstrukturen

Schnelle Übersicht

FH MÜNSTER
University of Applied Sciences

- `if` .. `else` - **Anweisung**: Selektion oder Verzweigung von Anweisungen
- `switch` .. `case`: **Fallunterscheidungen** mit Schlüsselwörtern `break` und `default`.
- `while`, `for`, `do` .. `while` - **Schleifen**: Wiederholung von Anweisungen mit Schlüsselwort `continue` für den Sprung ans Schleifenende und `break` für die komplette Beendung der Schleife.
- `goto` – **Anweisung**: Absoluter Sprung. ACHTUNG: Sollte nur in sehr ausgewählten Fällen angewendet werden! Führt zur Verlust von Übersichtlichkeit und kann zu schweren Fehlern führen!

Wie in C, daher für uns nichts neues!

Beispiel

Berechne Fakultät (Konsolen Ein- Ausgabe)

Beispiel:

```
#include <iostream>
using namespace std;

int main() {
      cout << "Fakultät berechnen. Zahl >= 0? :";
      int n;
      cin >> n; // warte auf Eingabe eines int-Wertes in der Konsole

      unsigned long fak {1L};

      for(int i=1; i<= n; ++i)
      {
        fak*=i;
      }
      cout << n << "! = " << fak << endl;

      return 0;
}
```

Kontrollstrukturen

Bereichsbasierte for-Schleifen (engl. range-based for-loop)

Eine Sequenz wie beispielsweise ein Daten-Feld wie z.B. `int feld[10]` oder ein Daten-Container wie z.B. `std::vector<int> container`

Variable, durch die auf den aktuellen Wert aus *bereich* im jeweiligen Schleifendurchlauf zugegriffen werden kann

Datentyp eines Elementes von *bereich*

```
for(Typ variable : bereich)
{
  // mache etwas mit variable
}
```

seit C++11

Bereichsbasierte Schleifen sind hilfreich, wenn nach und nach auf alle Elemente eines Bereiches zugegriffen werden soll. Sie ersetzen aber nicht die aus C bekannten "normalen" for-Schleifen.

Beispiel

Berechne Summe

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  const int n {3};
  int bereich[n];
  cout << "Gebe " << n << " Zahlen ein:" << endl;
  for(int i=0; i<n; i++)
  {
    cin >> bereich[i];
  }
  int summe = 0;
  for(int variable: bereich)
  {
    summe += variable;
  }
  cout << "Die Summe ist " << summe << endl;
  return 0;
}
```

FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de