Technisches Programmieren in C++

Konventionen

Karl Riedling
Institut für Sensor- und Aktuatorsysteme

Konventionen

- Struktur eines C++-Programms
- Übersetzung eines C++-Programms
- Sprachelemente
- Begriffe

Konventionen

- **Struktur eines C++-Programms**
- Übersetzung eines C++-Programms
- Sprachelemente
- Begriffe

Struktur eines C++-Programms

- C++-Programm:
  - eine oder mehrere *Übersetzungseinheiten*
  - mit jeweils einer beliebigen Anzahl von *Funktionen.*
- *Übersetzungseinheit*
  = *Quell-Programmdatei* + *Header-Dateien*
- *Header-Dateien*:
  - über `#include`-Präprozessordirektiven eingebettet
  - Funktionsdeklarationen, Definitionen von Konstanten oder Objekttypen usw.

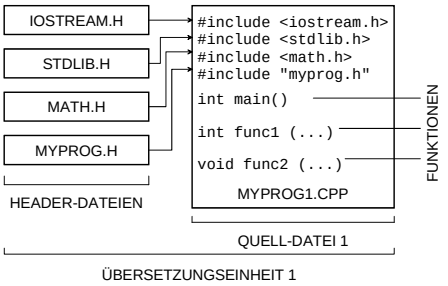
Struktur eines C++-Programms

- Dateitypen:
  - C++-Quell-Programmdateien: "CC", "CPP" oder "CXX";
  - Header-Dateien für C- oder C++-Programme: "H".
- C++-Programm = zumindest *eine* Funktion.
- Genau eine der Funktionen des Programms muss den Namen `main` haben (Ausnahme: Programme für *Microsoft Windows*: `WinMain`).

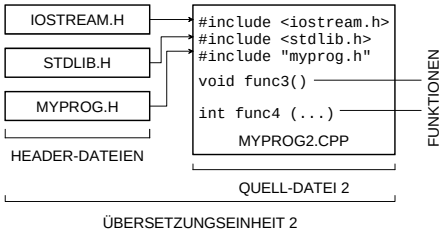
Struktur eines C++-Programms

- Die Funktion `main` (oder `WinMain`) wird nach den von den Laufzeit-Bibliotheksroutinen des C++-Compilers durchgeführten Initialisierungen aufgerufen;
- Rückkehr aus `main` (oder `WinMain`) = Beendigung des Programms.

Struktur eines C++-Programms



Struktur eines C++-Programms



Konventionen

- Struktur eines C++-Programms
- **Übersetzung eines C++-Programms**
- Sprachelemente
- Begriffe

Übersetzung

- Umsetzung des Zeichensatzes des Quellprogramms in den internen Zeichensatz.
- Zusammenhängen von Fortsetzungszeilen (Line Splicing mit "\").
- Umsetzung in *Tokens* (Kommentare ↯ Leerschritte; Aufteilung in Programm-Elemente und *White Space*).
- Vorverarbeitung (*Preprocessing*): Einbinden von Hilfsdateien über `#include`-Direktiven, Auflösung von Makros, ... ↯ "übersetzbare Einheit".
- Umsetzung des Quell-Zeichensatzes in den Ausführungs-Zeichensatz.

Übersetzung

- Zusammenhängen von Zeichenketten (*Strings*), die nur durch *White Space* voneinander getrennt sind.
- Umsetzung in den *Object-Code* (mit syntaktischer und semantischer Prüfung des Quellcodes).
- Zusammenbinden verschiedener Objekt-Module und Bibliotheksfunktionen zu einem ausführbaren Programm.

Konventionen

- Struktur eines C++-Programms
- Übersetzung eines C++-Programms
- **Sprachelemente**
- Begriffe

Konventionen

- Sprachelemente
  - ***Tokens***
  - Kommentare
  - Namen (*Identifiers*)
  - Schlüsselworte (*Keywords*)
  - Operatoren
  - Befehle
  - Konstanten

Tokens

- *Tokens* = kleinste Elemente eines C++-Programms:
  - Namen (*Identifier*);
  - Schlüsselworte (*Keywords*);
  - Numerische Konstanten und Textkonstanten (*Literals*);
  - Operatoren;
  - Trennzeichen.
- *Tokens* werden durch Leerräume ("*White Space*") oder Zeichen, die für ein bestimmtes *Token* nicht gültig sind, getrennt.

Tokens

- *White Space*:
  - Leerzeichen (*Spaces* — ASCII-Code `0x20` (20 Hex));
  - Horizontale und vertikale Tabulatoren (`0x09` und `0x0b`);
  - Zeilenvorschübe (`0x0d` und `0x0a`);
  - Seitenvorschübe (`0x0c`);
  - Kommentare.
- Beispiel:
  `y = sin(x);`
  "`y`", "`=`", "`sin`", "`(`", "`x`", "`)`" und "`;`" sind *Tokens*

Konventionen

- Sprachelemente
  - *Tokens*
  - **Kommentare**
  - Namen (*Identifiers*)
  - Schlüsselworte (*Keywords*)
  - Operatoren
  - Befehle (*Statements*)
  - Konstanten

Kommentare

- Kommentare werden vom Compiler (auch vom Präprozessor!) ignoriert.
- Zwei verschiedene Formen von Kommentaren:
  - "Klassischer" ANSI-C-Stil;
  - Einzelzeilen-Kommentare (*Single Line Comments*).

Kommentare

- "Klassischer" ANSI-C-Stil:
  - Kommentar zwischen "`/*`" und "`*/`" "eingeklammert":
    `/* Das ist ein Kommentar */`
  - ANSI-C-Kommentare dürfen sich über eine beliebige Zahl von Zeilen erstrecken;
  - Der Kommentar endet beim ersten "`*/`" nach dem einleitenden "`/*`" (deshalb keine geschachtelten Kommentare!).

Kommentare

- Einzelzeilen-Kommentare (*Single Line Comments*):
  - Kommentar beginnt mit "`//`" und endet mit dem Zeilenende (sofern nicht *unmittelbar* vor dem Zeilenvorschubs-Zeichen ein "\" steht):
    ```
    // Das ist auch ein Kommentar
    ```
- "`/*`", "`*/`" und "`//`" innerhalb von Zeichenketten (*Strings*) werden *nicht* als Kommentar-Steuerzeichen interpretiert.

Konventionen

- Sprachelemente
  - *Tokens*
  - Kommentare
  - **Namen (*Identifiers*)**
  - Schlüsselworte (*Keywords*)
  - Operatoren
  - Befehle (*Statements*)
  - Konstanten

Namen (*Identifiers*)

- Die folgenden Sprachelemente werden mit *Namen* bezeichnet:
  - Objekte oder Variable;
  - Klassen, Strukturen und *Unions*;
  - Aufzählungs-Typen;
  - Elemente einer Klasse, Struktur, *Union* oder Aufzählungs-Type;

Namen (Identifiers)

- Die folgenden Sprachelemente werden mit Namen bezeichnet (Fortsetzung):
  - Funktionen (auch Klassenelement-Funktionen);
  - Benutzerdefinierte Typen (`typedef`);
  - Sprungmarken;
  - Makros;
  - Argumente von Makros.

Namen (*Identifiers*)

- Namen dürfen aus folgenden Zeichen bestehen:
  ```
  _ a b c … x y z
  A B C … X Y Z
  0 1 2 … 8 9
  ```
- Namen dürfen *nicht* mit einer Ziffer (`0` … `9`) *beginnen*.
- Groß- und Kleinbuchstaben gelten als *verschieden* ("`DateiName`" ist nicht gleich "`Dateiname`" — *case sensitivity*).
- Namen dürfen nicht identisch zu Schlüsselworten sein.

Namen (*Identifiers*)

- Konventionen (zum Teil übernommen in ANSI-C-Norm):
  - Namen von Variablen, Funktionen und anderen vom *Compiler* aufzulösenden Einheiten (vorwiegend) in Kleinbuchstaben:
    `i`; `sin(x)`; `DateiName`; `datei_name`
  - Namen, die vom *Präprozessor* aufgelöst werden (also Namen von Makros, mit `#define` definierten Konstanten u.ä.), zur Gänze in Großbuchstaben:
    `PROGRAMM_VERSION`; `ZEILEN_PRO_SEITE`

Namen (Identifiers)

- Konventionen (Fortsetzung):
  - *Ungarische Notation* (nach dem Microsoft-Programmierer Charles Simonyi): Dem eigentlichen (Variablen-)Namen werden ein oder zwei Zeichen vorangestellt, die den Typ der Variablen angeben, z.B.:
    - `nZahl` — "n" für Integer.
    - `szname` — "`sz`" für String, Zero delimited
    - `fnMyFunc` — "`fn`" für Funktion(s-Zeiger).

Namen (Identifiers)

- "__" (zwei "_") oder "_" und ein Großbuchstabe am Anfang eines Namens sind für C++-Implementierungen reserviert.
- "_", gefolgt von einem Kleinbuchstaben, am Anfang eines Namens vermeiden für Namen, die über das aktuelle Modul hinaus gültig sind (z.B. globale Variable oder Funktionen) — Portabilitätsprobleme!

Konventionen

- Sprachelemente
  - *Tokens*
  - Kommentare
  - Namen (Identifiers)
  - **Schlüsselworte (Keywords)**
  - Operatoren
  - Befehle (Statements)
  - Konstanten

Schlüsselworte (*Keywords*)

- Schlüsselworte: Sind reserviert; dürfen nicht als Namen für Objekte oder Funktionen verwendet werden.
- Zusätzlich: implementierungsspezifische Schlüsselworte.

Schlüsselworte (*Keywords*)

`asm`	`double`	`new`	`switch`
`auto`	`else`	`operator`	`template`
`break`	`enum`	`private`	`this`
`case`	`extern`	`protected`	`throw`
`catch`	`float`	`public`	`try`
`char`	`for`	`register`	`typedef`
`class`	`friend`	`return`	`union`
`const`	`goto`	`short`	`unsigned`
`continue`	`if`	`signed`	`virtual`
`default`	`inline`	`sizeof`	`void`
`delete`	`int`	`static`	`volatile`
`do`	`long`	`struct`	`while`

Konventionen

- Sprachelemente
  - *Tokens*
  - Kommentare
  - Namen (*Identifiers*)
  - Schlüsselworte (*Keywords*)
  - **Operatoren**
  - Befehle (*Statements*)
  - Konstanten

Operatoren

- Operatoren werden nach *Präzedenz* gruppiert (siehe Tabelle im Skriptum).
- In einem Ausdruck werden Operatoren in der Reihenfolge abnehmender Präzedenz ausgewertet.

Konventionen

- Sprachelemente
  - *Tokens*
  - Kommentare
  - Namen (*Identifiers*)
  - Schlüsselworte (*Keywords*)
  - Operatoren
  - **Befehle (*Statements*)**
  - Konstanten

Befehle (*Statements*):

- Befehl (*Statement*): ein oder mehrere *Ausdrücke* (*Expressions*), verknüpft über Operatoren.
- Befehle sind in der Regel mit einem Strichpunkt (";") abgeschlossen:

```
a = b + c;
if (x > 0)
  i = 1;
else
  i = 0;
```


Befehle (*Statements*):

- *Block* oder *Verbundbefehl*:
  - Beliebige Anzahl von Befehlen durch geschwungene Klammern ("{ }") zusammengefasst.
  - Wird so wie ein einziger Befehl behandelt.
  - Befehle *innerhalb* eines Blocks sind *immer* mit Strichpunkten abzuschließen (auch der letzte!).

Befehle (*Statements*):

- *Block* oder *Verbundbefehl*:

```
if (x > 0)
{
  i = 1;
  j = -1;
}
else
{
  i = 0;
  k = 1;
}
```


Befehle (*Statements*):

- Der "Körper" einer Funktion ist *immer* ein Block in geschwungenen Klammern, auch wenn er nur einen einzigen Befehl enthält:

```
int faktorielle (unsigned n)
{
  int nfak;
  for (nfak = 1; n > 1; n--)
    nfak *= n;        // nfak = nfak * n;
  return nfak;
}
```

Konventionen

- Sprachelemente
  - *Tokens*
  - Kommentare
  - Namen (*Identifiers*)
  - Schlüsselworte (*Keywords*)
  - Operatoren
  - Befehle (*Statements*)
  - **Konstanten**

Konventionen

- Sprachelemente
  - Konstanten
    - **Ganzzahlige (*Integer*-) Konstanten**
    - Zeichenkonstanten (*Character*-Konstanten)
    - Gleitkommakonstanten
    - Zeichenketten- (*String*-) Konstanten

Ganzzahlige Konstanten

- Ganzzahlige *(Integer-)* Konstanten beginnen grundsätzlich immer mit einer Ziffer:
  - Dezimale Konstanten: Beginnen mit einer der Ziffern zwischen `1` und `9` (*nicht* `0`); können alle Ziffern von `0` bis `9` enthalten.
  - Oktale Konstanten: Beginnen immer mit "`0`" und enthalten nur Ziffern zwischen `0` und `7`.
  - Hexadezimale Konstanten: Beginnen mit "`0x`" oder "`0X`" und können neben den Ziffern von `0` bis `9` die hexadezimalen Ziffern "a" bis "`f`" (oder "A" bis "F") enthalten.

Ganzzahlige Konstanten

- Beispiele:

Dezimal:	**Oktal:**	**Hexadezimal:**
`123`	`0173`	`0x7b`
`32767`	`077777`	`0x7fff`
`2147483647`	`017777777777`	`0x7fffffff`

Ganzzahlige Konstanten

- Optional: Nachgestelltes "u" oder "U" = Konstante als `unsigned` (nicht vorzeichenbehaftet) zu interpretieren.
- Konstanten, die als `long integer` darzustellen sind, wird "`l`" oder "`L`" nachgestellt.
- "U" und "L" dürfen auch kombiniert werden:
  `unsigned long l = 4000000000UL;`

Ganzzahlige Konstanten

- Beispiel: 16-Bit-Implementierung mit
  - (*Short*) *Integer* = 16 Bit
  - *Long Integer* = 32 Bit
- Wertebereich einer *Short Integer*: 2^{16} Werte
  - –32767 … 32767 (*Signed Integer*) oder
  - 0 … 65535 (*Unsigned Integer*).
- "`40000`": Keine gültige (vorzeichenbehaftete) (*Short*) *Integer*-Konstante (außerhalb des vorzeichenbehafteten Wertevorrats gelegen), aber eine gültige *Unsigned Integer*-Konstante ("`40000u`" oder "`40000U`").

Ganzzahlige Konstanten

- In C++ bei Bedarf automatische Konversion in das Datenformat der Zielvariablen;
- Vorzeichen und/oder Wert der Konstanten können dabei aber verändert werden.

Konventionen

- Sprachelemente
  - Konstanten
    - Ganzzahlige (*Integer*-) Konstanten
    - **Zeichenkonstanten (*Character*-Konstanten)**
    - Gleitkommakonstanten
    - Zeichenketten- (*String*-) Konstanten

Zeichenkonstanten

- Quell-Zeichensatz (*Source Character Set*) — Zeichensatz, in dem das Programm geschrieben wurde;
- Ausführungs-Zeichensatz (*Execution Character Set*) — Zeichensatz, in dem das Programm mit seinen BenützerInnen spricht.
- Zeichenkonstanten gehören immer dem Quell-Zeichensatz an, stellen aber Zeichen des Ausführungs-Zeichensatzes dar.

Zeichenkonstanten

- Typen von Zeichenkonstanten:
  - "Gewöhnliche" Zeichenkonstanten: Immer vom Typ `char`:

    ```
    char ch = 'a';
    ```

  - "Weite" Zeichenkonstanten (*Wide character constants*, z.B. *Unicode*) : Vom Typ `wchar_t`:

    ```
    wchar_t unicode_char = L 'µ';
    ```

    (Präfix "L" !)

Zeichenkonstanten

- Darstellung von Zeichenkonstanten:
  - Ein oder mehrere Zeichen des Quell-Zeichensatzes mit Ausnahme von "`'`", "`\`" und des Zeilenvorschubs in einfachen Anführungszeichen:

    `'a'`, `'X'`, `'Ü'`
  - Einfache *Escape*-Sequenzen:

Escape-Sequenzen (Teil 1)

Zeichen	ASCII-Bezeichnung	ASCII-Wert	Escape-Sequenz
Zeilenvorschub	NL oder LF	0x0a	`\n`
Horizontaler Tabulator	HT	0x09	`\t`
Vertikaler Tabulator	VT	0x0b	`\v`
Rückschritt	BS	0x08	`\b`
Zeilenrücklauf	CR	0x0d	`\r`
Seitenvorschub	FF	0x0c	`\f`
Alarm	BEL	0x07	`\a`
Null-Zeichen	NUL	0x00	`\0`

Escape-Sequenzen (Teil 2)

Zeichen	ASCII-Bezeichnung	ASCII-Wert	Escape-Sequenz
Backslash	\	0x5c	`\\`
Fragezeichen	?	0x3f	`\?`
Einfaches Anführungszeichen	'	0x27	`\'`
Doppeltes Anführungszeichen	"	0x22	`\"`

Zeichenkonstanten

- Darstellung von Zeichenkonstanten:
  - Oktale Zeichenkonstanten: *Backslash* + 1 – 3 oktale Ziffern; enden mit dem ersten Zeichen, das keine oktale Zahl ist, oder spätestens nach drei Zeichen:

```
'\377'          // ASCII-Wert 255
```

  - Hexadezimale Zeichenkonstanten: "`\x`" + *beliebige* Anzahl hexadezimaler Ziffern; enden mit dem ersten Zeichen, das keine hexadezimale Zahl ist:

```
'\xff'          // ASCII-Wert 255
'\x00ff'        // gleiche Konstante
```


Konventionen

- Sprachelemente
  - Konstanten
    - Ganzzahlige (*Integer*-) Konstanten
    - Zeichenkonstanten (*Character*-Konstanten)
    - **Gleitkommakonstanten**
    - Zeichenketten- (*String*-) Konstanten

Gleitkommakonstanten

- Mindestens eine der folgenden Komponenten:
  - Dezimalpunkt;
  - Dezimalstellen;
  - Exponent ("e" oder "E", gefolgt von einer ganzzahligen Konstanten).

Gleitkommakonstanten

- Beispiele:

```
123.        // gültig
123.456     // gültig
0.456       // gültig
.456        // gültig
123e1       // gültig (= 1230.)
3.456e-6    // gültig (= 0.000003456)
```

- "123" ist *keine* Gleitkommakonstante!

Gleitkommakonstanten

- Datentyp von Gleitkommakonstanten:
  - Standardmäßig: `double`;
  - Nachstellen von "`f`" oder "`F`" ↯ Konstanten vom Typ `float`;
  - Nachstellen von "`l`" oder "`L`" ↯ Konstanten vom Typ `long double`.

Konventionen

- Sprachelemente
  - Konstanten
    - Ganzzahlige (*Integer*-) Konstanten
    - Zeichenkonstanten (*Character*-Konstanten)
    - Gleitkommakonstanten
    - **Zeichenketten- (*String*-) Konstanten**

Zeichenketten-Konstanten

- *String*konstanten: Ein oder mehrere Zeichen des Quellzeichensatzes oder Escape-Sequenzen in doppelten Anführungszeichen ("); immer mit einem Null-Byte (`NUL`, '`\0`') abgeschlossen.
- Darstellung von *Strings*:
  - Felder vom Typ `char[`*n*`]`, (*n* = Anzahl der Zeichen im String + 1 (Null-Byte!)):

```
char msg[14] = "Hello, world!";
  // Feldgrößen-Angabe ist unnötig
char msg[] = "Hello, world!";
  // Feldgröße ist 14 (13 + 1)
```


Zeichenketten-Konstanten

- Darstellung von *Strings*:
  - Felder vom Typ `wchar_t[`*n*`]`:

```
wchar_t WideString[] =
  L"\x1234\x2345";
    // WideString ist 6 Bytes lang
    // (4+2)
```


Zeichenketten-Konstanten

- Aneinanderstoßende Zeichenketten werden zusammengehängt:

```
"12" "34"   // äquivalent zu "1234"
```

- Gilt auch, wenn die *Strings* durch Zeilenvorschübe getrennt sind:

```
"The quick brown fox jumps "
"over the lazy white dog."
```

- Alternative: *Line Splicing*:

```
"The quick brown fox jumps \
over the lazy white dog."
```


Konventionen

- Struktur eines C++-Programms
- Übersetzung eines C++-Programms
- Sprachelemente
- **Begriffe**

Konventionen

- Begriffe
  - **Deklarationen und Definitionen**
  - Gültigkeitsbereich (*Scope*)
  - Dateiübergreifende Gültigkeit (*Linkage*)
  - Verknüpfung mit Standard-C-Funktionen
  - Speicherklassen
  - `const`, `volatile` und `inline`

Deklarationen und Definitionen

- Deklarationen:
  - Führen *Namen* und *Typen* in einem Programm ein, ohne notwendigerweise ein zugehöriges Objekt oder eine Funktion zu erzeugen.
  - Vielfach erfolgen *Deklarationen* im Rahmen von *Definitionen*.
  - Deklarationen, die *nicht* gleichzeitig auch Definitionen sind, dürfen auch öfter als einmal in einem Programm(modul) vorkommen, sofern sie einander nicht widersprechen (z.B.: Funktions-*Prototypen*, `typedef`-Befehle).

Deklarationen und Definitionen

- Definitionen:
  - Enthalten Informationen, die es dem Compiler ermöglichen, Speicherplatz für *Objekte* zu reservieren oder Programmcode für *Funktionen* zu generieren.
  - Ein Objekt oder eine Funktion muss *genau einmal definiert* werden.

Konventionen

- Begriffe
  - Deklarationen und Definitionen
  - **Gültigkeitsbereich (*Scope*)**
  - Dateiübergreifende Gültigkeit (*Linkage*)
  - Verknüpfung mit Standard-C-Funktionen
  - Speicherklassen
  - `const`, `volatile` und `inline`

Gültigkeitsbereich (*Scope*)

- Gültigkeitsbereich eines (Objekts- oder Funktions-) Namens;
- Gibt den Bereich des Programms an, innerhalb dessen auf dieses Objekt unter diesem Namen zugegriffen werden kann.

Gültigkeitsbereich (*Scope*)

- Lokale Gültigkeit:
  - Namen, die *innerhalb* eines Blocks deklariert werden;
  - Formale Argumente von Funktionen: werden so behandelt, als wären sie im äußersten Block der Funktion deklariert worden.
- Funktions-Gültigkeit:
  - Nur für Sprungmarken (*Labels*).

Gültigkeitsbereich (*Scope*)

- Datei-Gültigkeit:
  - Namen, die *außerhalb* aller Blöcke oder Klassen deklariert wurden;
  - Wenn sie nicht *statische* Objekte angeben, werden Namen mit Datei-Gültigkeit oft auch als *globale* Namen bezeichnet.
- Klassen-Gültigkeit (*Class Scope*):
  - Namen der Elemente einer Klasse, die nur mit Hilfe eines Elementauswahl-Operators ("`.`" oder "`->`"), angewandt auf ein Objekt (oder einen Zeiger auf ein Objekt) dieser Klasse angesprochen werden können.

Gültigkeitsbereich (*Scope*)

- Klassen-Gültigkeit (*Class Scope*):
  - Beispiel:

```
class Point
{
  int x;
  int y;
};
```

  - x und y haben Klassen-Gültigkeit.

Gültigkeitsbereich (*Scope*)

- *Prototypen*-Gültigkeit:
  - Gilt nur für die (optionalen) Namen in einem Funktions-*Prototyp*.
  - Beispiel:

```
char *strcpy (char *szDest, const char
*szSource);
```

`szDest` und `szSource` haben *Prototypen*-Gültigkeit.

Gültigkeitsbereich (*Scope*)

- Namen gelten unmittelbar *nach* ihrer Deklaration, aber noch *vor* ihrer allfälligen Initialisierung als deklariert.
- Namen können *verborgen* werden, wenn sie innerhalb eines eingeschlossenen Blocks neu deklariert werden:

Gültigkeitsbereich (*Scope*)

```
#include <iostream.h>            // Definition von "cout"
int i = 1;                       // Datei-Gültigkeit
void MyFunc (int i)
{
  cout << "i = " << i << "\n";   // gibt Argument i aus
  {
    int i = 2;
    cout << "i = " << i << "\n";     // gibt aus "i = 2"
    {
      int i = 3;
      cout << "i = " << i << "\n";   // gibt aus "i = 3"
    }
    cout << "i = " << i << "\n";     // gibt aus "i = 2"
  }
  cout << "i = " << i << "\n";   // gibt Argument i aus
}
```


Gültigkeitsbereich (*Scope*)

```
// Fortsetzung

void MyFunc1 ()
{
  cout << "i = " << i << "\n";
    // gibt den Wert der globalen Variable i aus ("i = 1")
}
```


Konventionen

- Begriffe
  - Deklarationen und Definitionen
  - Gültigkeitsbereich (*Scope*)
  - **Dateiübergreifende Gültigkeit (*Linkage*)**
  - Verknüpfung mit Standard-C-Funktionen
  - Speicherklassen
  - `const`, `volatile` und `inline`

Linkage

- Dateiübergreifende Gültigkeit (*Linkage*) betrifft Programme, die aus mehreren *Übersetzungseinheiten* (*Modulen; Translation Units*) bestehen:
  - Interne (lokale)
  - Externe
  - Keine

Linkage

- Interne (lokale) *Linkage*:
  - Namen in einer Übersetzungseinheit sind unabhängig von allfälligen identischen Namen in irgendeiner anderen Übersetzungseinheit.
  - Der selbe Name kann in verschiedenen Modulen verschiedene Objekte (auch verschiedenen Typs) kennzeichnen.

Linkage

- Interne (lokale) *Linkage*:
  - Innerhalb eines Blocks deklarierte Namen sind lokal (außer, sie wurden mit dem Schlüsselwort "`extern`" deklariert).
  - Auch mit Datei-Gültigkeit (d.h., außerhalb aller Blöcke) mit dem Schlüsselwort "`static`" deklarierte Objekte sind lokal:

```
static int i;
```


Linkage

- Externe *Linkage*:
  - Externe Objekte sind solche mit Datei-Gültigkeit, die *nicht* mit dem Schlüsselwort "`static`" deklariert wurden.
  - Externe Objekte, die *nicht* in der aktuellen Übersetzungseinheit *definiert* werden, können optional mit dem Schlüsselwort `extern` *deklariert* werden:

Linkage

```
// Datei A:
int i = 3;
  // globale Variable deklariert und definiert
// Datei B:
#include <iostream.h>           // für cout
extern int i;   // deklariert, aber nicht definiert
...
{
  cout << "i = " << i << "\n";  // gibt "i = 3" aus
}
```


Linkage

aktuelles Modul:	andere Module:	Bemerkungen

`int i;`	`int i;`
	`extern int i;`
	`int i = 3;`
		Initialisierung nur in *einer* Übersetzungseinheit;
		`i` wird in der *Object*-Datei als *communal* registriert

`extern int i;`	`int i;`
	`extern int i;`
	`int i = 3;`
		Initialisierung nur in *einer* Übersetzungseinheit;
		`i` wird in der *Object*-Datei als *external* registriert

`int i = 3;`	`int i;` `extern int i;`	`i` wird in der *Object*-Datei als *public* registriert

Linkage

- Keine dateiübergreifende Gültigkeit:
  - Solche Objekte sind einmalig vorhanden.
  - Auf sie kann von außerhalb ihres Gültigkeits-bereiches nicht zugegriffen werden.
  - Beispiele:
    - Alle Objekte mit lokaler Gültigkeit, sofern sie nicht mit dem Schlüsselwort "`extern`" deklariert wurden;
    - Funktions-Argumente;
    - Aufzählungen;
    - `typedef`-Namen.

Konventionen

- Begriffe
  - Deklarationen und Definitionen
  - Gültigkeitsbereich (*Scope*)
  - Dateiübergreifende Gültigkeit (*Linkage*)
  - **Verknüpfung mit Standard-C-Funktionen**
  - Speicherklassen
  - `const`, `volatile` und `inline`

Verknüpfung mit Standard-C

- Originale (globale) Namen werden in übersetzten *Object*-Dateien umgesetzt ("*dekoriert*").
- C++ verwendet dafür andere Konventionen als Standard-C:
  - Standard-C: *Dekoration* globaler Namen durch Voranstellen von "_".
  - C++: Type, allfällige Klassenzugehörigkeit sowie Anzahl und Typ der Argumente von Funktionen werden in codierter Form in den *dekorierten* Namen aufgenommen.

Verknüpfung mit Standard-C

- Nicht-C++-Funktionen müssen als solche deklariert werden, um aus C++ aufgerufen werden zu können.
- Deklaration von Nicht-C++-Funktionen mit "`extern "C"`":

```
extern "C" int printf (const char *fmt, ...);
```


Verknüpfung mit Standard-C

- Gemeinsame Deklaration von Nicht-C++-Funktionen oder Objekten:

```
extern "C"
{
  void *malloc (size_t size);
  void free (void *ptr);
  int errno;
}
```


Konventionen

- Begriffe
  - Deklarationen und Definitionen
  - Gültigkeitsbereich (*Scope*)
  - Dateiübergreifende Gültigkeit (*Linkage*)
  - Verknüpfung mit Standard-C-Funktionen
  - **Speicherklassen**
  - `const`, `volatile` und `inline`

Speicherklassen

- Speicherklassen bestimmen Lebensdauer, dateiübergreifende Gültigkeit (*Linkage*) und Behandlung von Objekten und Variablen:
  - Automatisch
  - Statisch
  - Extern
  - Register

Speicherklassen

- Automatische Speicherklasse:
  - Objekte mit automatischer Speicherklasse sind lokal für jeden Aufruf des Blocks, innerhalb dessen sie definiert wurden.
  - Bei paralleler (*multithreaded*) oder rekursiver Ausführung eines Blocks wird garantiert, dass ihnen für jeden Aufruf des Blocks individuelle Speicherplätze zugeordnet werden.
  - Automatische Daten werden in der Regel am Stack abgelegt.

Speicherklassen

- Automatische Speicherklasse:
  - Alle Objekte, die innerhalb eines Blocks definiert wurden, haben standardmäßig die Speicherklasse `auto`, sofern sie nicht unter Verwendung von `extern` oder `static` deklariert wurden.
  - Die Verwendung von "`auto`" ist optional.
  - Automatische Objekte haben keine dateiübergreifende Gültigkeit; sie existieren bis zum Ende des Blocks, innerhalb dessen sie deklariert wurden.

Speicherklassen

- Statische Speicherklasse:
  - Objekte und Variable, die unter Verwendung des Schlüsselwortes `static` deklariert wurden, werden auf fix festgelegten Speicherplätzen im Arbeitsspeicher abgelegt.
  - Statische Variable und Objekte bestehen vom Zeitpunkt ihrer Definition an bis zum Ende des Programms.

Speicherklassen

- Statische Speicherklasse:
  - Bei paralleler (*multithreaded*) oder rekursiver Ausführung eines Blocks wird garantiert, dass für jeden Aufruf des Blocks auf *denselben* Speicherplatz und daher auf *denselben* Zustand oder Wert des Objekts zugegriffen wird.
  - Objekte und Variable, die *außerhalb* aller Blöcke definiert wurden, haben statische Speicherklasse.

Speicherklassen

- Statische Speicherklasse:
  - Sie haben standardmäßig *externe* Gültigkeit, außer, wenn sie mit dem Schlüsselwort `static` deklariert wurden.
  - In diesem Fall sind sie nur innerhalb der aktuellen Übersetzungseinheit gültig.
  - Die Verwendung des Schlüsselwortes `static` ist auch für die Deklaration von Funktionen zulässig.
  - Diese sind dann ebenfalls nur innerhalb der aktuellen Übersetzungseinheit gültig.

Speicherklassen

- Externe Speicherklasse:
  - Objekte oder Funktionen, die unter Verwendung des Schlüsselwortes `extern` deklariert werden, deklarieren ein Objekt, das in einer anderen Quelldatei oder in einem höheren Block mit externer Gültigkeit definiert wurde.

Speicherklassen

- Register-Speicherklasse:
  - Variable, die unter Verwendung des Schlüsselwortes `register` deklariert wurden, werden in einem Register der CPU abgelegt.
  - Sie verhalten sich ansonsten exakt wie automatische Variable.
  - Die Speicherklasse `register` ist nur für lokale Variable und für die formalen Argumente einer Funktion zulässig.

Speicherklassen

- Initialisierung in Abhängigkeit von der Speicherklasse:
  - Lokale *automatische* Objekte: *immer*, wenn der Programmablauf ihre Definition erreicht.
  - Lokale *statische* Objekte: wenn der Programmablauf *erstmals* ihre Definition erreicht.

Speicherklassen

```
#include <iostream.h>   // für cout
void myfunc ()
{
  static int j = 1;
  int k = 1;
  cout << "j = " << j << ", k = " << k << "\n";
  j++; k++;
}
int main ()
{
  for (int i = 0; i < 5; i++)
    myfunc ();
}
```


Speicherklassen

- Ausgabe des Beispielprogramms:

```
j = 1, k = 1
j = 2, k = 1
j = 3, k = 1
j = 4, k = 1
j = 5, k = 1
```


Speicherklassen

- In C++ können (im Gegensatz zu C) Definitionen von Variablen und Objekten *an jeder beliebigen Stelle eines Blocks* (also nicht nur vor der ersten ausführbaren Anweisung) stehen.
- Externe Objekte dürfen nur in einer einzigen Übersetzungseinheit des Programms initialisiert (= *definiert*), aber beliebig oft *deklariert* werden.

Konventionen

- Begriffe
  - Deklarationen und Definitionen
  - Gültigkeitsbereich (*Scope*)
  - Dateiübergreifende Gültigkeit (*Linkage*)
  - Verknüpfung mit Standard-C-Funktionen
  - Speicherklassen
  - **`const`, `volatile` und `inline`**

Konventionen

- Begriffe
  - `const`, `volatile` und `inline`
    - **`const`**
    - `volatile`
    - `inline`

`const`

- Behandlung eines Objekts als Konstante:
  - Deklaration echter Konstanten:

```
const double Pi = 3.141592654;
```

  - Schutz von per Referenz an eine Funktion übergebenen Argumenten vor unbeabsichtigten Änderungen:

```
char *strcpy (char *target, const char
*source);
```


`const`

- In beiden Fällen verhindert der Compiler eine Veränderung dieser Objekte:

```
Pi = 6.28;
  // erzeugt eine Compiler-
  // Fehlermeldung
```


`const`

- Objekte mit Datei-Gültigkeit, die als `const` und *nicht* als `extern` deklariert wurden, sind lokal für ihre Übersetzungseinheit:

```
const int lokal = 2;
   // nur in aktueller Übersetzungseinheit
   // zugänglich
extern const int global = 3;
   // auch aus anderen Übersetzungs-
   // einheiten zugänglich
```


Konventionen

- Begriffe
  - `const`, `volatile` und `inline`
    - `const`
    - **`volatile`**
    - `inline`

volatile

- Das Schlüsselwort `volatile` gibt an, dass das betreffende Objekt während der Ausführung des Programms asynchron verändert werden kann.
  - Beispiel: eine Semaphor-Variable, die von einem Interrupt-Handler aus gesetzt wird):

```
volatile int ReadyFlag;
```

- Dieses Attribut dient in erster Linie dazu, gewisse Optimierungen zu unterbinden.
- Eine mit `volatile` deklarierte Variable wird bei jedem Zugriff neu aus dem Arbeitsspeicher geladen.

Konventionen

- Begriffe
  - `const`, `volatile` und `inline`
    - `const`
    - `volatile`
    - **`inline`**

inline

- Das Schlüsselwort "`inline`" darf nur bei der Deklaration von Funktionen verwendet werden.
- Der Code solcher Funktionen wird für *jeden* Aufruf der Funktion *separat* vorgesehen.
- Vorteil: Vermeidung des Overheads eines Funktionsaufrufs.
- Funktionell ist eine `inline`-Funktion äquivalent zu einem Makro; aber:
  - Typenprüfung der Argumente und des Ergebnisses;
  - Nebenwirkungen wie bei Makros werden vermieden.

inline

- Funktionen, die als `inline` deklariert wurden, sind immer lokal für ihre Übersetzungseinheit.
- `inline`-Funktionen sind sinnvoll für einfache Anwendungen (z.B. in Zugriffsfunktionen von Klassen), wo der Code-Overhead eines Funktionsaufrufs größer wäre als die Expansion des `inline`-Codes.
- Typische `inline`-Funktion:

```
inline int square (int i)
{
  return i*i;
}
```