



# Programmierpraktikum Technische Informatik (C++)



# Überblick

- Grundlagen C++
- Toolchain Teil 2
- Hinweise zu Fehlermeldungen



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

# Grundlagen C++



# Inhalt

## ■ Funktionen

# Inhalt

- Funktionen
- Komplexe Datentypen

# Inhalt

- Funktionen
- Komplexe Datentypen
- Templates



# Inhalt

- Funktionen
- Komplexe Datentypen
- Templates
- Standardbibliothek

# Inhalt

- Funktionen
- Komplexe Datentypen
- Templates
- Standardbibliothek
- Verschiedenes



# Komplexe Datentypen

- Komplexe Datentypen sind alle nicht in die Sprache eingebauten Datentypen

# Komplexe Datentypen

- Komplexe Datentypen sind alle nicht in die Sprache eingebauten Datentypen
- Bereits bekannt: `std::string`
  - Teil der C++-Standardbibliothek

# Komplexe Datentypen

- Komplexe Datentypen sind alle nicht in die Sprache eingebauten Datentypen
- Bereits bekannt: `std::string`
  - Teil der C++-Standardbibliothek
- Bessere Unterstützung komplexer Datentypen wichtigstes Merkmal von C++ gegenüber C

# Komplexe Datentypen

- Komplexe Datentypen sind alle nicht in die Sprache eingebauten Datentypen
- Bereits bekannt: `std::string`
  - Teil der C++-Standardbibliothek
- Bessere Unterstützung komplexer Datentypen wichtigstes Merkmal von C++ gegenüber C
- Komplexe Datentypen können neben Daten auch Funktionen besitzen, sog. Memberfunktionen

# Memberfunktionen

- Wie normale Funktionen, nur mit anderer Aufrufsyntax

# Memberfunktionen

- Wie normale Funktionen, nur mit anderer Aufrufsyntax
- Memberfunktionen werden mit *object.function(arguments)* aufgerufen:

```
std::string foo("Hello");  
foo.append(" world");  
std::string("Goodbye World").swap(foo);
```

## Memberfunktionen

- Wie normale Funktionen, nur mit anderer Aufrufsyntax
- Memberfunktionen werden mit *object.function(arguments)* aufgerufen:

```
std::string foo("Hello");  
foo.append(" world");  
std::string("Goodbye World").swap(foo);
```

- Memberfunktionen benötigen zum Aufruf eine Instanz des Typs
  - `foo.resize(5)` ist korrekt, `std::string.resize(5)` nicht

# Operatoren

- Operatoren (+, -, \*, ...) können in C++ für komplexe Datentypen umdefiniert werden
  - “Operator Overloading”



# Operatoren

- Operatoren (+, -, \*, ...) können in C++ für komplexe Datentypen umdefiniert werden
  - “Operator Overloading”
- Stringkonkatenation mit + aus der letzten Veranstaltung bekannt
- `std::string` überlädt außerdem: Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=`, `>=`

# Operatoren

- Operatoren (+, -, \*, ...) können in C++ für komplexe Datentypen umdefiniert werden
  - “Operator Overloading”
- Stringkonkatenation mit + aus der letzten Veranstaltung bekannt
- `std::string` überlädt außerdem: Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Wichtig:** Zeichenkettenlitterale (z.B. `"foo"`) sind nicht vom Typ `std::string`
  - Bei Arbeiten nur mit Zeichenkettenlitteralen müssen C-Funktionen verwendet werden, Operatoren funktionieren nicht
    - Keine Garantie, ob `"foo" == "foo"` wahr oder falsch ergibt

# Operatoren

- Operatoren (+, -, \*, ...) können in C++ für komplexe Datentypen umdefiniert werden
  - "Operator Overloading"
- Stringkonkatenation mit + aus der letzten Veranstaltung bekannt
- `std::string` überlädt außerdem: Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Wichtig:** Zeichenkettenlitterale (z.B. `"foo"`) sind nicht vom Typ `std::string`
  - Bei Arbeiten nur mit Zeichenkettenlitteralen müssen C-Funktionen verwendet werden, Operatoren funktionieren nicht
    - Keine Garantie, ob `"foo" == "foo"` wahr oder falsch ergibt
  - Können implizit in `std::string` umgewandelt werden
    - `std::string("foo") == "foo"` funktioniert, da Operator von `std::string` vorgegeben wird

## Kopien von komplexen Datentypen

```
std::string foo("hello world");  
std::string bar = foo;  
bar.resize(5);
```

- C++ ermöglicht Entwicklern festzulegen, wie komplexe Datentypen bei Zuweisung kopiert werden

## Kopien von komplexen Datentypen

```
std::string foo("hello world");  
std::string bar = foo;  
bar.resize(5);
```

- C++ ermöglicht Entwicklern festzulegen, wie komplexe Datentypen bei Zuweisung kopiert werden
- Kopierverhalten für die meisten C++-Bibliotheken: “Deep Copy”
  - Kopie des Objektes erstellt vom Original unabhängiges Objekt
  - Wichtiger Unterschied zu Java: Zuweisung erstellt in Java für komplexe Datentypen lediglich eine zweite Referenz auf Originalobjekt

## Kopien von komplexen Datentypen

```
std::string foo("hello world");
std::string bar = foo;
bar.resize(5);
```

- C++ ermöglicht Entwicklern festzulegen, wie komplexe Datentypen bei Zuweisung kopiert werden
- Kopierverhalten für die meisten C++-Bibliotheken: “Deep Copy”
  - Kopie des Objektes erstellt vom Original unabhängiges Objekt
  - Wichtiger Unterschied zu Java: Zuweisung erstellt in Java für komplexe Datentypen lediglich eine zweite Referenz auf Originalobjekt
- bar hat keine Verbindung zu foo, bar.resize() ändert die Größe von foo nicht

## Kopien von komplexen Datentypen

```
std::string foo("hello world");
std::string bar = foo;
bar.resize(5);
```

- C++ ermöglicht Entwicklern festzulegen, wie komplexe Datentypen bei Zuweisung kopiert werden
- Kopierverhalten für die meisten C++-Bibliotheken: “Deep Copy”
  - Kopie des Objektes erstellt vom Original unabhängiges Objekt
  - Wichtiger Unterschied zu Java: Zuweisung erstellt in Java für komplexe Datentypen lediglich eine zweite Referenz auf Originalobjekt
- bar hat keine Verbindung zu foo, bar.resize() ändert die Größe von foo nicht
- Erleichtert Nachverfolgbarkeit von Code
  - Identität von Objekten klarer festgelegt

## Kopien von komplexen Datentypen

```
std::string foo("hello world");
std::string bar = foo;
bar.resize(5);
```

- C++ ermöglicht Entwicklern festzulegen, wie komplexe Datentypen bei Zuweisung kopiert werden
- Kopierverhalten für die meisten C++-Bibliotheken: “Deep Copy”
  - Kopie des Objektes erstellt vom Original unabhängiges Objekt
  - Wichtiger Unterschied zu Java: Zuweisung erstellt in Java für komplexe Datentypen lediglich eine zweite Referenz auf Originalobjekt
- bar hat keine Verbindung zu foo, bar.resize() ändert die Größe von foo nicht
- Erleichtert Nachverfolgbarkeit von Code
  - Identität von Objekten klarer festgelegt
- Birgt Performancerisiken, unnötige Kopien erzeugen Laufzeitoverhead



# Überladene Funktionen

- C++ erlaubt Funktionsüberladung
- Verschiedene Funktionen mit gleichem Namen

# Überladene Funktionen

- C++ erlaubt Funktionsüberladung
- Verschiedene Funktionen mit gleichem Namen

```
int    min(int    a, int    b) { return a < b ? a : b; }  
float  min(float  a, float  b) { return a < b ? a : b; }
```

# Überladene Funktionen

- C++ erlaubt Funktionsüberladung
- Verschiedene Funktionen mit gleichem Namen

```
int    min(int    a, int    b) { return a < b ? a : b; }  
float  min(float  a, float  b) { return a < b ? a : b; }
```

- Compiler muss die Funktionen beim Aufruf unterscheiden können
  - Unterscheidung in den Parametertypen
  - Unterscheidung nur im Rückgabetyt reicht **nicht**

# Überladene Funktionen

- C++ erlaubt Funktionsüberladung
- Verschiedene Funktionen mit gleichem Namen

```
int min(int a, int b) { return a < b ? a : b; }  
float min(float a, float b) { return a < b ? a : b; }
```

- Compiler muss die Funktionen beim Aufruf unterscheiden können
  - Unterscheidung in den Parametertypen
  - Unterscheidung nur im Rückgabetyt reicht **nicht**
- Überladene Funktionen sollten in der Regel ähnliche Operationen implementieren
  - Kann sonst leicht zu Verwirrung führen

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5);
```

```
foo(5, true);
```

```
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5); ← foo(5, false, "bar")
```

```
foo(5, true);
```

```
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5);
```

```
foo(5, true); ← foo(5, true, "bar")
```

```
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5);
```

```
foo(5, true);
```

```
foo(5, false, "foobar") ← foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter



## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5);
```

```
foo(5, true);
```

```
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter
- Reduziert Codeduplikation

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5);
```

```
foo(5, true);
```

```
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter
- Reduziert Codeduplikation
- Parameter mit Defaultwerten nur am Ende der Parameterliste
  - Illegal: `void foo(int=5, bool)`

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
```

```
foo(5);
```

```
foo(5, true);
```

```
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter
- Reduziert Codeduplikation
- Parameter mit Defaultwerten nur am Ende der Parameterliste
  - Illegal: `void foo(int=5, bool)`
- Defaultparameter können zu `ambiguous-overload`-Fehlern führen

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));  
void foo(int);  
foo(5);  
foo(5, true);  
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter
- Reduziert Codeduplikation
- Parameter mit Defaultwerten nur am Ende der Parameterliste
  - Illegal: `void foo(int=5, bool)`
- Defaultparameter können zu `ambiguous-overload`-Fehlern führen

## Defaultparameter für Funktionen

```
void foo(int, bool=false, std::string=std::string("bar"));
void foo(int);
foo(5); ← call of overloaded 'foo(int)' is ambiguous
foo(5, true);
foo(5, false, "foobar")
```

- Anders als C und Java unterstützt C++ Defaultwerte für Parameter
- Reduziert Codeduplikation
- Parameter mit Defaultwerten nur am Ende der Parameterliste
  - Illegal: `void foo(int=5, bool)`
- Defaultparameter können zu ambiguous-overload-Fehlern führen

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufrufenden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufzufinden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);
```

```
foo(5);
```

## Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufrufenden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);
```

```
foo(5); ← error: too few arguments to function 'void foo(int, int)'
```



# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufrufenden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);  
void foo(int x, int y = 0);  
foo(5);
```

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufzufinden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);
```

```
void foo(int x, int y = 0);
```

```
foo(5); ← Ok, äquivalent zu foo(5, 0);
```

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufrufenden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);
void foo(int x, int y = 0);
foo(5);
void foo(int x, int y);
```

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufrufenden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);  
void foo(int x, int y = 0);  
foo(5);
```

```
void foo(int x, int y); ← Ok, redeclariert foo ohne Defaultparameter hinzuzufügen
```

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufzufinden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);  
void foo(int x, int y = 0);  
foo(5);  
void foo(int x, int y);  
void foo(int x, int y = 0);
```

# Deklaration von Funktionen mit Defaultparametern

- Defaultwerte für Parameter werden an der **aufzufinden** Codestelle eingefügt
  - Defaultwerte müssen dem Compiler bekannt sein
  - Defaultwerte gehören in die Deklaration der Funktion
  - Bei mehrfacher Deklaration einer Funktion darf nur eine Defaultparameter erhalten

```
void foo(int x, int y);  
void foo(int x, int y = 0);  
foo(5);  
void foo(int x, int y);  
void foo(int x, int y = 0);
```

error: default argument given for parameter 2 of 'void foo(int, int)'  
after previous specification in 'void foo(int, int)'

# const

- Schlüsselwort `const` zur Definition konstanter Variablen
- `const` `T` beschreibt einen konstanten Typ

```
const int foo = 5;
```

# const

- Schlüsselwort `const` zur Definition konstanter Variablen
- `const T` beschreibt einen konstanten Typ
- Konstante Variablen nach Initialisierung nicht mehr modifizierbar

```
const int foo = 5;
```

```
foo = 2; ← error: assignment of read-only variable 'foo'
```

```
++foo; ← error: increment of read-only variable 'foo'
```



# const

- Schlüsselwort `const` zur Definition konstanter Variablen
- `const T` beschreibt einen konstanten Typ
- Konstante Variablen nach Initialisierung nicht mehr modifizierbar
- `const T` und `T const` sind äquivalent
  - Später noch wichtig

```
const int foo = 5;  
foo = 2;  
++foo;  
int const bar = 5;
```

# Verwendung von const

- Nichtmodifizierende Memberfunktionen dürfen aufgerufen werden

## Verwendung von const

- Nichtmodifizierende Memberfunktionen dürfen aufgerufen werden
  - `const` `std::string` `x("foobar");`
  - Legal: `x.size()`
  - Illegal: `x.resize(3)`

## Verwendung von const

- Nichtmodifizierende Memberfunktionen dürfen aufgerufen werden
  - `const std::string x("foobar");`
  - Legal: `x.size()`
  - Illegal: `x.resize(3)`
- `const` stellt sicher, dass konstante Variablen nicht modifiziert werden
  - Hilft anderen Entwicklern beim Verstehen des Quellcodes

## Verwendung von const

- Nichtmodifizierende Memberfunktionen dürfen aufgerufen werden
  - `const` `std::string` `x("foobar");`
  - Legal: `x.size()`
  - Illegal: `x.resize(3)`
- `const` stellt sicher, dass konstante Variablen nicht modifiziert werden
  - Hilft anderen Entwicklern beim Verstehen des Quellcodes

Wann immer möglich `const` verwenden!

## Call by Value

```
void foo(int x) { x = 5; }  
int main()  
{  
    int bar=10;  
    foo(bar);  
    std::cout<<bar<<std::endl;  
}
```

- Was ist die Ausgabe dieses Programmes?

## Call by Value

```
void foo(int x) { x = 5; }  
int main()  
{  
    int bar=10;  
    foo(bar);  
    std::cout<<bar<<std::endl;  
}
```

- Was ist die Ausgabe dieses Programmes?
  - Antwort: 10

## Call by Value

```
void foo(int x) { x = 5; }  
int main()  
{  
    int bar=10;  
    foo(bar);  
    std::cout<<bar<<std::endl;  
}
```

- Was ist die Ausgabe dieses Programmes?
  - Antwort: 10
- Funktionsargumente werden “by value” übergeben
- Die Funktion operiert auf einer Kopie der übergebenen Daten



## Referenzen

```
int    foo = 5;
int&   bar = foo;
int    baz = 10;
std::cout<<foo<<"", "<<bar<<", "<<baz<<"\n";
bar = baz;
std::cout<<foo<<"", "<<bar<<", "<<baz<<"\n";
foo += 5;
std::cout<<foo<<"", "<<bar<<", "<<baz<<"\n";
```

- Referenztypen: *type-name*& (bsp: `int&`)
- Referenzen sind Verweise auf andere Objekte

## Referenzen

```
int    foo = 5;
int&   bar = foo;
int    baz = 10;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
bar = baz;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
foo += 5;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
```

- Referenztypen: *type-name*& (bsp: `int&`)
- Referenzen sind Verweise auf andere Objekte
- Werden bei Initialisierung an zu referenzierendes Objekt gebunden
  - Illegal: `int& bar;`

## Referenzen

```
int    foo = 5;
int&   bar = foo;
int    baz = 10;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
bar = baz;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
foo += 5;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
```

- Referenztypen: *type-name*& (bsp: `int&`)
- Referenzen sind Verweise auf andere Objekte
- Werden bei Initialisierung an zu referenzierendes Objekt gebunden
  - Illegal: `int& bar;`
- Verhalten sich wie das referenzierte Objekt
- Äquivalent zu C-Pointern mit Dereferenzierung bei jedem Zugriff

## Referenzen

```
int    foo = 5;
int&   bar = foo;
int    baz = 10;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n"; ← 5, 5, 10
bar = baz;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
foo += 5;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
```

- Referenztypen: *type-name*& (bsp: `int&`)
- Referenzen sind Verweise auf andere Objekte
- Werden bei Initialisierung an zu referenzierendes Objekt gebunden
  - Illegal: `int& bar;`
- Verhalten sich wie das referenzierte Objekt
- Äquivalent zu C-Pointern mit Dereferenzierung bei jedem Zugriff

## Referenzen

```
int    foo = 5;
int&   bar = foo;
int    baz = 10;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
bar = baz;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n"; ← 10, 10, 10
foo += 5;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
```

- Referenztypen: *type-name*& (bsp: `int&`)
- Referenzen sind Verweise auf andere Objekte
- Werden bei Initialisierung an zu referenzierendes Objekt gebunden
  - Illegal: `int& bar;`
- Verhalten sich wie das referenzierte Objekt
- Äquivalent zu C-Pointern mit Dereferenzierung bei jedem Zugriff

## Referenzen

```
int    foo = 5;
int&   bar = foo;
int    baz = 10;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
bar = baz;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n";
foo += 5;
std::cout<<foo<<"", "<<bar<<"", "<<baz<<"\n"; ← 15, 15, 10
```

- Referenztypen: *type-name*& (bsp: `int&`)
- Referenzen sind Verweise auf andere Objekte
- Werden bei Initialisierung an zu referenzierendes Objekt gebunden
  - Illegal: `int& bar;`
- Verhalten sich wie das referenzierte Objekt
- Äquivalent zu C-Pointern mit Dereferenzierung bei jedem Zugriff

## lvalues

- `int& x = 5;` nicht möglich
- Referenzen können nur an lvalues gebunden werden

## Ivalues

- `int& x = 5;` nicht möglich
- Referenzen können nur an Ivalues gebunden werden
- Ivalues: Können **links** eines Zuweisungsoperators stehen
  - Meistens Variablen
  - Simplifizierte Darstellung, Wahrheit etwas komplizierter



## Ivalues

- `int& x = 5;` nicht möglich
- Referenzen können nur an Ivalues gebunden werden
- Ivalues: Können **links** eines Zuweisungsoperators stehen
  - Meistens Variablen
  - Simplifizierte Darstellung, Wahrheit etwas komplizierter
- Alternative Vereinfachung:
  - Ivalues sind Objekte mit einem Namen

## Ivalues

- `int& x = 5;` nicht möglich
- Referenzen können nur an Ivalues gebunden werden
- Ivalues: Können **links** eines Zuweisungsoperators stehen
  - Meistens Variablen
  - Simplifizierte Darstellung, Wahrheit etwas komplizierter
- Alternative Vereinfachung:
  - Ivalues sind Objekte mit einem Namen
- Klassische Ivalues:
  - Variablen
  - Funktionsparameter

## rvalues

- Können nur **rechts** eines Zuweisungsoperators stehen
  - Temporäre Objekte
  - Wahrheit auch hier komplizierter

## rvalues

- Können nur **rechts** eines Zuweisungsoperators stehen
  - Temporäre Objekte
  - Wahrheit auch hier komplizierter
- x ist typischer lvalue, 5 rvalue

## rvalues

- Können nur **rechts** eines Zuweisungsoperators stehen
  - Temporäre Objekte
  - Wahrheit auch hier komplizierter
- `x` ist typischer lvalue, 5 rvalue
- Referenzen auf konstante Objekte können auch an rvalues gebunden werden
  - `const int& x = 5;` ist erlaubt
  - Compiler erzeugt für diesen Code eine Variable vom Typ `int`, die den Wert 5 erhält und von `x` referenziert wird
  - Lebenszeit dieser Variablen entspricht der von `x`

## rvalues

- Können nur **rechts** eines Zuweisungsoperators stehen
  - Temporäre Objekte
  - Wahrheit auch hier komplizierter
- x ist typischer lvalue, 5 rvalue
- Referenzen auf konstante Objekte können auch an rvalues gebunden werden
  - `const int& x = 5;` ist erlaubt
  - Compiler erzeugt für diesen Code eine Variable vom Typ `int`, die den Wert 5 erhält und von x referenziert wird
  - Lebenszeit dieser Variablen entspricht der von x
- Typische rvalues:
  - Temporäre Objekte (`std::string()`, Funktionsrückgabewerte)
  - Literale (5, `"hello world"`, `true`)

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);
int& bar();
void baz(const std::string&);
```

```
int x = 0;
int z = foo(x);
x = x + 5;
baz(std::string(5, ' '));
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);
int& bar();
void baz(const std::string&);
```

```
int x = 0;
int z = foo(x);
x = x + 5;
baz(std::string(5, ' '));
bar() = 5;
```

lvalue



## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);
int& bar();
void baz(const std::string&);
```

```
int x = 0;
int z = foo(x);
x = x + 5;
baz(std::string(5, ' '));
bar() = 5;
```

rvalue

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

lvalue

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);
int& bar();
void baz(const std::string&);
```

```
int x = 0;
int z = foo(x);
x = x + 5;
baz(std::string(5, ' '));
bar() = 5;
```

rvalue

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);
int& bar();
void baz(const std::string&);
```

```
int x = 0;
int z = foo(x);
x = x + 5;
baz(std::string(5, ' '));
bar() = 5;
```

rvalue



## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);
int& bar();
void baz(const std::string&);
```

```
int x = 0;
int z = foo(x);
x = x + 5;
baz(std::string(5, ' '));
bar() = 5;
```

rvalue

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

## Quiz

Ist das markierte Objekt bzw. der Rückgabewert der markierten Operation ein lvalue oder ein rvalue?

```
int foo(int x);  
int& bar();  
void baz(const std::string&);
```

```
int x = 0;  
int z = foo(x);  
x = x + 5;  
baz(std::string(5, ' '));  
bar() = 5;
```

lvalue

# Call by Reference

- Referenzparameter erlauben Funktionen, auf den übergebenen Variablen zu operieren

```
void foo(int& x) { x = 5; }  
int main() {  
    int bar=10;  
    foo(bar);  
    std::cout<<bar<<std::endl;  
}
```

# Call by Reference

- Referenzparameter erlauben Funktionen, auf den übergebenen Variablen zu operieren

```
void foo(int& x) { x = 5; }
int main() {
    int bar=10;
    foo(bar);
    std::cout<<bar<<std::endl;
}
```

- Ausgabe: 5

## Call by Reference

- Referenzparameter erlauben Funktionen, auf den übergebenen Variablen zu operieren

```
void foo(int& x) { x = 5; }
int main() {
    int bar=10;
    foo(bar);
    std::cout<<bar<<std::endl;
}
```

- Ausgabe: 5
- Kann nur mit lvalues aufgerufen werden
- `int qux(const int& x)` auch mit rvalues aufrufbar

## Wahl der Aufrufkonvention

- Call by Reference, wenn Funktion Argumente modifizieren sollen
- Call by Value, wenn die Funktion eine Kopie der Daten benötigt
- Call by Value bei kleinen Objekten
  - Anlegen einer Referenz für primitive Datentypen nicht billiger als das Erstellen einer Kopie
  - Zugriff über Referenz möglicherweise teurer als bei einer Kopie
- Ansonsten Call by Const-Reference, um die Kosten für eine Kopie zu vermeiden



## Wahl der Aufrufkonvention

- Call by Reference, wenn Funktion Argumente modifizieren sollen
- Call by Value, wenn die Funktion eine Kopie der Daten benötigt
- Call by Value bei kleinen Objekten
  - Anlegen einer Referenz für primitive Datentypen nicht billiger als das Erstellen einer Kopie
  - Zugriff über Referenz möglicherweise teurer als bei einer Kopie
- Ansonsten Call by Const-Reference, um die Kosten für eine Kopie zu vermeiden

```
void foo(std::string x) {}  
void bar(const std::string& x) {}  
  
std::string x("this is a teststring");  
for(long long i = 0; i < 1000000000; ++i) foo(x);  
for(long long i = 0; i < 1000000000; ++i) bar(x);
```

## Wahl der Aufrufkonvention

- Call by Reference, wenn Funktion Argumente modifizieren sollen
- Call by Value, wenn die Funktion eine Kopie der Daten benötigt
- Call by Value bei kleinen Objekten
  - Anlegen einer Referenz für primitive Datentypen nicht billiger als das Erstellen einer Kopie
  - Zugriff über Referenz möglicherweise teurer als bei einer Kopie
- Ansonsten Call by Const-Reference, um die Kosten für eine Kopie zu vermeiden

```
void foo(std::string x) {}  
void bar(const std::string& x) {}
```

```
std::string x("this is a teststring");  
for(long long i = 0; i < 1000000000; ++i) foo(x); ← 0m19.052s  
for(long long i = 0; i < 1000000000; ++i) bar(x);
```

## Wahl der Aufrufkonvention

- Call by Reference, wenn Funktion Argumente modifizieren sollen
- Call by Value, wenn die Funktion eine Kopie der Daten benötigt
- Call by Value bei kleinen Objekten
  - Anlegen einer Referenz für primitive Datentypen nicht billiger als das Erstellen einer Kopie
  - Zugriff über Referenz möglicherweise teurer als bei einer Kopie
- Ansonsten Call by Const-Reference, um die Kosten für eine Kopie zu vermeiden

```
void foo(std::string x) {}
void bar(const std::string& x) {}
```

```
std::string x("this is a teststring");
for(long long i = 0; i < 1000000000; ++i) foo(x); ← 0m19.052s
for(long long i = 0; i < 1000000000; ++i) bar(x); ← 0m1.671s
```

## Rückgaben und Referenzen

```
int& foo()
{
    int x = 5;
    return x;
}
```

- Lebenszeit von `x` mit Verlassen von `foo` beendet
- `foo` gibt Referenz auf ein Objekt zurück, das nicht mehr existiert
- Undefiniertes Verhalten nach dem C++-Standard
- Daher: Rückgabe immer by Value

## Rückgaben und Referenzen

```
int& foo()
{
    int x = 5; ← warning: reference to local variable 'x' returned
    return x;
}
```

- Lebenszeit von `x` mit Verlassen von `foo` beendet
- `foo` gibt Referenz auf ein Objekt zurück, das nicht mehr existiert
- Undefiniertes Verhalten nach dem C++-Standard
- Daher: Rückgabe immer by Value

## Rückgaben und Referenzen

```
int& foo()  
{  
    int x = 5; ← warning: reference to local variable 'x' returned  
    return x;  
}
```

- Lebenszeit von `x` mit Verlassen von `foo` beendet
- `foo` gibt Referenz auf ein Objekt zurück, das nicht mehr existiert
- Undefiniertes Verhalten nach dem C++-Standard
- Daher: Rückgabe immer by Value
- Compiler kann das Anlegen einer Kopie für den Rückgabewert vermeiden
  - Return Value Optimization (RVO), deckt gängige Fälle ab

## Rückgaben und Referenzen

```
int& foo()
{
    int x = 5; ← warning: reference to local variable 'x' returned
    return x;
}
```

- Lebenszeit von `x` mit Verlassen von `foo` beendet
- `foo` gibt Referenz auf ein Objekt zurück, das nicht mehr existiert
- Undefiniertes Verhalten nach dem C++-Standard
- Daher: Rückgabe immer by Value
- Compiler kann das Anlegen einer Kopie für den Rückgabewert vermeiden
  - Return Value Optimization (RVO), deckt gängige Fälle ab

Rückgabe von lokalen Variablen immer by Value, nie by Reference!

# Templates

- Templates sind eine weitere wichtige Neuerung in C++ gegenüber C
- Ähnlich zu Generics in Java, allerdings bedeutend mächtiger
  - Templatesystem von C++ ist turingvollständig
- Grundlage für einen Großteil der C++-Standardbibliothek



# Motivation für Templates

```
int    min(int    a, int    b) { return a < b ? a : b; }
long   min(long   a, long   b) { return a < b ? a : b; }
float  min(float  a, float  b) { return a < b ? a : b; }
...
```

- Häufig ist die selbe Operation für verschiedene Datentypen sinnvoll

## Motivation für Templates

```
int    min(int    a, int    b) { return a < b ? a : b; }
long   min(long   a, long   b) { return a < b ? a : b; }
float  min(float  a, float  b) { return a < b ? a : b; }
...
```

- Häufig ist die selbe Operation für verschiedene Datentypen sinnvoll
- In C muss die Funktion für jeden konkreten Datentyp einzeln geschrieben werden
  - Problematisch, da Wartung erschwert wird
  - Alternative: Macro, hat aber eigene Probleme, z.B. unbeabsichtigte Mehrfachauswertung von Ausdrücken

## Motivation für Templates

```
int    min(int    a, int    b) { return a < b ? a : b; }
long  min(long   a, long   b) { return a < b ? a : b; }
float min(float  a, float  b) { return a < b ? a : b; }
...
```

- Häufig ist die selbe Operation für verschiedene Datentypen sinnvoll
- In C muss die Funktion für jeden konkreten Datentyp einzeln geschrieben werden
  - Problematisch, da Wartung erschwert wird
  - Alternative: Macro, hat aber eigene Probleme,
    - z.B. unbeabsichtigte Mehrfachauswertung von Ausdrücken
- Templates erlauben es, eine generische Funktion zu schreiben, die beim Aufruf für den jeweiligen Datentyp konkretisiert wird

## Motivation für Templates

```
int    min(int    a, int    b) { return a < b ? a : b; }
long  min(long   a, long   b) { return a < b ? a : b; }
float min(float  a, float  b) { return a < b ? a : b; }
...
```

- Häufig ist die selbe Operation für verschiedene Datentypen sinnvoll
- In C muss die Funktion für jeden konkreten Datentyp einzeln geschrieben werden
  - Problematisch, da Wartung erschwert wird
  - Alternative: Macro, hat aber eigene Probleme, z.B. unbeabsichtigte Mehrfachauswertung von Ausdrücken
- Templates erlauben es, eine generische Funktion zu schreiben, die beim Aufruf für den jeweiligen Datentyp konkretisiert wird
- Standardbibliothek enthält Template `std::min`, kann als `std::min(a, b)` für beliebige Datentypen aufgerufen werden
  - Einschränkung: Typ von `a` und `b` muss für `std::min` identisch sein

# Template-Argumente

- Templates selber sind keine Funktionen/Datentypen
- Durch Spezifikation der Templateargumente werden Templates instanziiert

# Template-Argumente

- Templates selber sind keine Funktionen/Datentypen
- Durch Spezifikation der Templateargumente werden Templates instanziiert
- Templateargumente stehen in spitzen Klammern <> nach dem Namen der Template-Funktion/Template-Klasse
  - Bsp: `std::min<int>(a, b)`
  - Im Folgenden: *T* Platzhalter für Template-Argumente

## Template-Argumente II

- Template-Argumente können für Template-Funktionen oft vom Compiler implizit bestimmt werden (**deduziert**)
- Existiert Funktionsargument von Typ  $T$ , wird  $T$  auf den Typ des Parameters gesetzt
  - a und b vom Typ `int`:  
`std::min(a, b)` entspricht `std::min<int>(a, b)`

## Template-Argumente II

- Template-Argumente können für Template-Funktionen oft vom Compiler implizit bestimmt werden (**deduziert**)
- Existiert Funktionsargument von Typ  $T$ , wird  $T$  auf den Typ des Parameters gesetzt
  - a und b vom Typ `int`:  
`std::min(a, b)` entspricht `std::min<int>(a, b)`
- Gibt  $T$  den Typ mehrerer Funktionsargumente an, müssen diese für Deduktion den selben Typ haben
  - `std::min(1, 0.5)` kompiliert nicht, da  $T=int$  und  $T=double$  deduziert wird
  - Explizite Angabe der Template-Argumente notwendig (`std::min<double>(1, 0.5)` funktioniert)



## std::vector

```
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> collection = {1, 2, 3, 4, 5, 6, 7, 8};
    collection.emplace_back(9);
    for(std::size_t i = 0; i < collection.size(); ++i)
        std::cout << collection[i] << "\n";
    return 0;
}
```

- Die C++-Standardbibliothek enthält Container als Template-Klassen
  - Standard Template Library (STL) Teil der Standardbibliothek
  - Angabe von Templateargumenten wie bei Funktionen mit <>
- std::vector kapselt ein dynamisches Array
  - Entsprechung in Java: ArrayList
  - C verfügt über keine derartige Kapselung

## std::vector

```
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> collection = {1, 2, 3, 4, 5, 6, 7, 8};
    collection.emplace_back(9);
    for(std::size_t i = 0; i < collection.size(); ++i)
        std::cout << collection[i] << "\n";
    return 0;
}
```

- C++ bietet seit C++11 Syntax zur direkten Initialisierung von Containertypen
- {1, ..., 8} erzeugt eine `std::initializer_list<int>`
- `std::vector<T>` verfügt über Konstruktor zur Konstruktion aus `std::initializer_list<T>`

## std::vector

```
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> collection = {1, 2, 3, 4, 5, 6, 7, 8};
    collection.emplace_back(9);
    for(std::size_t i = 0; i < collection.size(); ++i)
        std::cout << collection[i] << "\n";
    return 0;
}
```

- Mit `myVec.emplace_back(T)` können weitere Elemente an das Ende des Vectors angehängt werden
  - Parameter: Argumente für den Konstruktor von *T*
  - Objekt wird direkt an Endposition konstruiert
- Größe des Vectors wird automatisch angepasst

## std::vector

```
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> collection = {1, 2, 3, 4, 5, 6, 7, 8};
    collection.emplace_back(9);
    for(std::size_t i = 0; i < collection.size(); ++i)
        std::cout << collection[i] << "\n";
    return 0;
}
```

- size\_t (eigentlich std::size\_t) ist Typ für Größenangaben und Indizes in C++
  - Groß genug, um alle möglichen Größen abbilden zu können
  - Vorzeichenlos. Vorzeichenbehaftete Entsprechung: ptrdiff\_t (oder std::ptrdiff\_t)
- .size() gibt die Anzahl der Elemente eines std::vector zurück

## std::vector

```
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> collection = {1, 2, 3, 4, 5, 6, 7, 8};
    collection.emplace_back(9);
    for(std::size_t i = 0; i < collection.size(); ++i)
        std::cout << collection[i] << "\n";
    return 0;
}
```

- Mit `myVec[i]` kann auf Elemente des Vectors zugegriffen werden

# Iteratoren

- Häufig: Iterieren über alle Elemente eines Containers

# Iteratoren

- Häufig: Iterieren über alle Elemente eines Containers
- Wünschenswert: Vom Containertyp unabhängiges Interface

# Iteratoren

- Häufig: Iterieren über alle Elemente eines Containers
- Wünschenswert: Vom Containertyp unabhängiges Interface
- Iteratoren sind Zeiger auf Elemente eines Containers
- Können auf vorhergehendes und nächstes Element verschoben werden



# Iteratoren

- Häufig: Iterieren über alle Elemente eines Containers
- Wünschenswert: Vom Containertyp unabhängiges Interface
- Iteratoren sind Zeiger auf Elemente eines Containers
- Können auf vorhergehendes und nächstes Element verschoben werden
- Syntax an die C-Syntax für Pointer angelehnt
- Iteratoren können im Gegensatz zu Pointern auch über komplexe Datenstrukturen wie verkettete Listen iterieren

## Verwendung von Iteratoren

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for (std::vector<std::string>::const_iterator iter = myVec.begin();
     iter != myVec.end();
     ++iter)
    std::cout << *iter;
```

- *container*::const\_iterator ist Typ für Iteratoren, durch die Containerelemente **nicht** verändert werden können
- Sind Modifikationen gewünscht: *container*::iterator
- iterators können implizit in const\_iterators konvertiert werden
- Konvertierung von const\_iterator zu iterator nicht ohne weiteres möglich
  - Würde Const-Correctness verletzen

## Verwendung von Iteratoren

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};  
for(std::vector<std::string>::const_iterator iter = myVec.begin();  
    iter != myVec.end();  
    ++iter)  
    std::cout<< *iter;
```

- Die Methode `begin()` gibt Iterator zurück, der auf erstes Element des Containers zeigt
- Rückgabetyt hängt von der Constness des Containers ab:
  - `const_iterator`, wenn der Container `const` ist
  - ansonsten `iterator`

## Verwendung von Iteratoren

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};  
for(std::vector<std::string>::const_iterator iter = myVec.begin();  
    iter != myVec.end();  
    ++iter)  
    std::cout<< *iter;
```

- Die Methode `end()` gibt einen Iterator zurück, der **hinter** das letzte Element zeigt
- Zugriff auf referenziertes Element ist undefiniertes Verhalten
  - Häufig: Segmentation Fault
- Iteratoren können mit `==` und `!=` verglichen werden
  - Einige Iteratoren unterstützen auch `<`, `<=`, `>` und `>=`
- `iter != container.end()` bedeutet: Iteriere, solange der Iterator in den Containergrenzen liegt

## Verwendung von Iteratoren

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::vector<std::string>::const_iterator iter = myVec.begin();
    iter != myVec.end();
    ++iter)
    std::cout<< *iter;
```

- Inkrementoperator (++iter, iter++) schiebt Iteratoren ein Element weiter
- Analog schiebt der Dekrementoperator einen Iterator ein Element zurück
- Operatoren, die um mehrere Elemente verschieben (+, -, +=, -=) werden nicht von allen Iteratortypen unterstützt

## Verwendung von Iteratoren

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::vector<std::string>::const_iterator iter = myVec.begin();
    iter != myVec.end();
    ++iter)
    std::cout<< *iter;
```

- Zugriff auf das referenzierte Element des Containers geschieht über den Dereferenzierungsoperator *\*iter*
- Im Falle von komplexen Elementtypen Zugriff auf Memberfunktionen über (*\*iter*).*member*()
- Äquivalent, aber eleganter: *iter->member*()

## Verwendung von Iteratoren

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::vector<std::string>::const_iterator iter = myVec.begin();
    iter != myVec.end();
    ++iter)
    std::cout<< *iter;
```

- Achtung: Änderungen am Container können Iteratoren invalidieren
  - Änderungen in der Container-Struktur kritisch
  - Änderungen der Elemente selbst unkritisch
  - `myVec[1]="foo"` invalidiert nicht, `myVec.emplace_back("bar")` schon
- Regeln dafür später
- Für den Moment: Annahme, dass alle Operationen, die den Container verändern, invalidieren

# Range-based For

- Verwendung von Iteratoren sehr umständlich



## Range-based For

- Verwendung von Iteratoren sehr umständlich
- Seit C++11: Range-based For

# Range-based For

- Verwendung von Iteratoren sehr umständlich
- Seit C++11: Range-based For

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::string value: myVec)
    std::cout<< value;
```

## Range-based For

- Verwendung von Iteratoren sehr umständlich
- Seit C++11: Range-based For

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};  
for(std::string value: myVec)  
    std::cout<< value;
```

- Ermöglicht einfaches Iterieren über alle Container, die begin und end unterstützen
- Bis auf Erstellung einer lokalen Variable value äquivalent zu vorherigem Beispiel

## Range-based For

- Verwendung von Iteratoren sehr umständlich

- Seit C++11: Range-based For

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::string value: myVec)
    std::cout<< value;
```

- Ermöglicht einfaches Iterieren über alle Container, die `begin` und `end` unterstützen
- Bis auf Erstellung einer lokalen Variable `value` äquivalent zu vorherigem Beispiel
- Achtung: `value` ist jeweils Kopie des aktuellen Elements
  - Containerinhalt kann darüber nicht geändert werden
  - Unnötiger Laufzeitoverhead

## Range-based For

- Verwendung von Iteratoren sehr umständlich

- Seit C++11: Range-based For

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};  
for(std::string& value: myVec)  
    std::cout<< value;
```

- Verwendung von Referenzen vermeidet unnötige Kopien

## Range-based For

- Verwendung von Iteratoren sehr umständlich

- Seit C++11: Range-based For

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::string& value: myVec)
    std::cout<< value;
```

- Verwendung von Referenzen vermeidet unnötige Kopien
- In diesem Fall wäre `const std::string&` noch besser

## Range-based For

- Verwendung von Iteratoren sehr umständlich

- Seit C++11: Range-based For

```
std::vector<std::string> myVec = {"hello", "world", "!\n"};
for(std::string& value: myVec)
    std::cout<< value;
```

- Verwendung von Referenzen vermeidet unnötige Kopien
- In diesem Fall wäre `const std::string&` noch besser
- Iteration über Teilbereiche des Vectors leider immer noch umständlich
  - Elegante Ansätze in anderen Bibliotheken (später mehr)

## std::map

- `std::map` ist weiterer Container der Standardbibliothek



## std::map

- `std::map` ist weiterer Container der Standardbibliothek
- Geordneter Key/Value Container
  - Basiert auf einem balancierten binären Suchbaum
  - Entspricht dem Typ `TreeMap` in Java

## std::map

- `std::map` ist weiterer Container der Standardbibliothek
- Geordneter Key/Value Container
  - Basiert auf einem balancierten binären Suchbaum
  - Entspricht dem Typ `TreeMap` in Java
- Logarithmische Laufzeit für Einfügen, Löschen und Finden eines Schlüssels
- Iterieren über gesamten Container linear in der Anzahl der Elemente

## Beispiel std::map

```
#include <map>
std::map<std::string, int> dictionary = {{"Alice", 1}, {"Bob", 3}};
dictionary.insert({"Eve", 2});
std::map<std::string, int>::iterator bobIter = dictionary.find("Bob");
if(bobIter != dictionary.end())
    std::cout<<"Bob found\n";
for(std::map<std::string, int>::value_type& elem: dictionary)
    std::cout<<elem.first<<": " <<elem.second<<"\n";
```

- std::map benötigt zwei Typparameter
- Erster Parameter ist der Typ des Keys (hier: std::string)
- Zweiter Parameter gibt den Typ für Werte an (hier: int)

## Beispiel std::map

```
#include <map>
std::map<std::string, int> dictionary = {"Alice", 1}, {"Bob", 3};
dictionary.insert({"Eve", 2});
std::map<std::string, int>::iterator bobIter = dictionary.find("Bob");
if(bobIter != dictionary.end())
    std::cout<<"Bob found\n";
for(std::map<std::string, int>::value_type& elem: dictionary)
    std::cout<<elem.first<<": " <<elem.second<<"\n";
```

- Initialisierung über eine verschachtelte Initialisierungsliste möglich

## Beispiel std::map

```
#include <map>
std::map<std::string, int> dictionary = {{"Alice", 1}, {"Bob", 3}};
dictionary.insert({"Eve", 2});
std::map<std::string, int>::iterator bobIter = dictionary.find("Bob");
if(bobIter != dictionary.end())
    std::cout<<"Bob found\n";
for(std::map<std::string, int>::value_type& elem: dictionary)
    std::cout<<elem.first<<": " <<elem.second<<"\n";
```

- .insert(*item*) fügt neues Element ein
- Typ der Elemente ist std::map<K, V>::value\_type
- {"Eve", 2} verwendet Initialisierungssyntax, um das einzufügende Objekt zu erzeugen, ohne den Typ angeben zu müssen
  - Geht überall, wo der Zieltyp bekannt ist, betrifft return, Funktionsargumente, Zuweisungen
  - Compilerfehler, wenn verschiedene Zieltypen zur Auswahl stehen

## Beispiel std::map

```
#include <map>
std::map<std::string, int> dictionary = {{"Alice", 1}, {"Bob", 3}};
dictionary.insert({"Eve", 2});
std::map<std::string, int>::iterator bobIter = dictionary.find("Bob");
if (bobIter != dictionary.end())
    std::cout<<"Bob found\n";
for(std::map<std::string, int>::value_type& elem: dictionary)
    std::cout<<elem.first<<": " <<elem.second<<"\n";
```

- `.find(key)` sucht das zu einem Key gehörende Element
- Gibt iterator auf das Element zurück, falls es in der map existiert
- Ansonsten entspricht der Rückgabewert `dictionary.end()`

## Beispiel std::map

```
#include <map>
std::map<std::string, int> dictionary = {{"Alice", 1}, {"Bob", 3}};
dictionary.insert({"Eve", 2});
std::map<std::string, int>::iterator bobIter = dictionary.find("Bob");
if(bobIter != dictionary.end())
    std::cout<<"Bob found\n";
for(std::map<std::string, int>::value_type& elem: dictionary)
    std::cout<<elem.first<<": " <<elem.second<<"\n";
```

- Zugriff auf den zugehörigen **Wert** auch über dictionary[*key*]
  - **Aber:** Erstellt Element, falls es nicht existiert
  - Lokalisierung von Fehlern schwieriger
  - Kann nicht für `const` Maps verwendet werden

## Beispiel std::map

```
#include <map>
std::map<std::string, int> dictionary = {{"Alice", 1}, {"Bob", 3}};
dictionary.insert({"Eve", 2});
std::map<std::string, int>::iterator bobIter = dictionary.find("Bob");
if(bobIter != dictionary.end())
    std::cout<<"Bob found\n";
for(std::map<std::string, int>::value_type& elem: dictionary)
    std::cout<<elem.first<<": " <<elem.second<<"\n";
```

- Zugriff auf Key mit `element.first`
- Zugriff auf Value mit `element.second`



## std::tuple

- Bisher: Homogene Container (speichern nur Elemente gleichen Typs)

## std::tuple

- Bisher: Homogene Container (speichern nur Elemente gleichen Typs)
- Jetzt: Inhomogene Container: Tuple

## std::tuple

- Bisher: Homogene Container (speichern nur Elemente gleichen Typs)
- Jetzt: Inhomogene Container: Tuple
- Aufgrund der statischen Typisierung von C++:  
Länge und beteiligte Elementtypen zur Compilezeit festgelegt
- Beispiel: `std::tuple<std::string, int, double>`

## std::tuple

- Bisher: Homogene Container (speichern nur Elemente gleichen Typs)
- Jetzt: Inhomogene Container: Tuple
- Aufgrund der statischen Typisierung von C++:  
Länge und beteiligte Elementtypen zur Compilezeit festgelegt
- Beispiel: `std::tuple<std::string, int, double>`
- Zugriff auf Elemente mit `std::get<n>(tuple)`
  - `n` gibt den Index des Elements an

## std::tuple

- Bisher: Homogene Container (speichern nur Elemente gleichen Typs)
- Jetzt: Inhomogene Container: Tuple
- Aufgrund der statischen Typisierung von C++:  
Länge und beteiligte Elementtypen zur Compilezeit festgelegt
- Beispiel: `std::tuple<std::string, int, double>`
- Zugriff auf Elemente mit `std::get<n>(tuple)`
  - `n` gibt den Index des Elements an

```
#include <tuple>
int main()
{
    std::tuple<int, int, bool> bar(10, 5, true);
    std::get<2>(bar) = false;
    std::cout << std::get<0>(bar) << "\n";
}
```

# IO

- Für IO-Operationen bietet C++ IOStreams
- `std::cin`, `std::cout` sind IOStream-Instanzen
  - Konkret: `std::istream` bzw. `std::ostream`

# IO

- Für IO-Operationen bietet C++ IOStreams
- `std::cin`, `std::cout` sind IOStream-Instanzen
  - Konkret: `std::istream` bzw. `std::ostream`
- C++ stellt Streamtypen für verschiedene Ziele bereit
  - `std::fstream` für File-IO (Header: `fstream`)
  - `std::stringstream` zum Schreiben auf/Lesen von `std::strings` (Header: `sstream`)
  - Jeweils Spezialisierung von `std::iostream`
  - Erlaubt sowohl Lese- als auch Schreibzugriffe
- Ein- und Auslesen wie gehabt mit `<<` und `>>`

## In- und Outstreams

- Häufig wird ein Stream nur zum Lesen oder nur zum Schreiben verwendet



## In- und Outstreams

- Häufig wird ein Stream nur zum Lesen oder nur zum Schreiben verwendet
- Varianten von `std::fstream` und `std::stringstream`:
  - `std::ofstream` und `std::ostream` erlauben nur Schreibzugriffe
  - `std::ifstream` und `std::istream` analog für Lesezugriffe

## In- und Outstreams

- Häufig wird ein Stream nur zum Lesen oder nur zum Schreiben verwendet
- Varianten von `std::fstream` und `std::stringstream`:
  - `std::ofstream` und `std::ostream` erlauben nur Schreibzugriffe
  - `std::ifstream` und `std::istream` analog für Lesezugriffe
- Sind jeweils Spezialisierungen von `std::istream` bzw. `std::ostream`
  - Können als solche verwendet und an Funktionen übergeben werden
  - `std::iostream` spezialisiert sowohl `std::istream` als auch `std::ostream`

## Beispiel: Anzahl Zeilen in einer Datei zählen

```
unsigned int calcLineCount(std::istream& is)
{
    unsigned int lineCount = 0;
    std::string line;
    while(std::getline(is, line))
        ++lineCount;
    return lineCount;
}

int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream fs(arguments[1]);
    std::cout << calcLineCount(fs) << "\n";
    return 0;
}
```

## Beispiel: Anzahl Zeilen in einer Datei zählen

```
unsigned int calcLineCount(std::istream& is)
{
    unsigned int lineCount = 0;
    std::string line;
    while(std::getline(is, line))
        ++lineCount;
    return lineCount;
}

int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream fs(arguments[1]);
    std::cout << calcLineCount(fs) << "\n";
    return 0;
}
```

- Alternative Signatur für main
- Kommandozeilenargumente als `char`-Pointer (wie in C)
- Pointer erst später
- Aktuell: Ignorieren

## Beispiel: Anzahl Zeilen in einer Datei zählen

```
unsigned int calcLineCount(std::istream& is)
{
    unsigned int lineCount = 0;
    std::string line;
    while(std::getline(is, line))
        ++lineCount;
    return lineCount;
}

int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream fs(arguments[1]);
    std::cout << calcLineCount(fs) <<
    return 0;
}
```

- Alternative Signatur für main
- Kommandozeilenargumente als `char`-Pointer (wie in C)
- Pointer erst später
- Aktuell: Ignorieren

- `vector` besitzt Konstruktor aus zwei Iteratoren
- Befüllt `arguments` mit den Kommandozeilenargumenten
- `arguments[0]` ist Name des Executables

## Beispiel: Anzahl Zeilen in einer Datei zählen

```
unsigned int calcLineCount(std::istream& is)
{
    unsigned int lineCount = 0;
    std::string line;
    while(std::getline(is, line))
        ++lineCount;
    return lineCount;
}
```

Öffnet Datei arguments[1] zum Lesen

```
int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream fs(arguments[1]);
    std::cout << calcLineCount(fs) <<
    return 0;
}
```

- Alternative Signatur für main
- Kommandozeilenargumente als **char**-Pointer (wie in C)
- Pointer erst später
- Aktuell: Ignorieren

- vector besitzt Konstruktor aus zwei Iteratoren
- Befüllt arguments mit den Kommandozeilenargumenten
- arguments[0] ist Name des Executables

## Beispiel: Anzahl Zeilen in einer Datei zählen

```
unsigned int calcLineCount(std::istream& is)
{
    unsigned int lineCount = 0;
    std::string line;
    while(std::getline(is, line))
        ++lineCount;
    return lineCount;
}
```

ifstream kann als istream verwendet werden

```
int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream fs(arguments[1]);
    std::cout << calcLineCount(fs) << "\n";
    return 0;
}
```

## Beispiel: Anzahl Zeilen in einer Datei zählen

```
unsigned int calcLineCount(std::istream& is)
{
    unsigned int lineCount = 0;
    std::string line;
    while(std::getline(is, line))
        ++lineCount;
    return lineCount;
}
```

- Liest eine Zeile aus is in line
- Gibt **false** zurück, wenn das Dateiende erreicht ist, sonst **true**
- Typ des ersten Arguments ist istream
  - Kann auch von der Konsole oder stringstreams einlesen
- Allgemeine Form: `std::getline(<stream>, <string>, <delim>)`
- Liest bis zum nächsten Auftreten von <delim> ein
- <delim> wird nicht mit eingelesen

stream verwendet werden



# Kopieren von Dateien

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>
int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream src(arguments[1]);
    std::ofstream dst(arguments[2]);
    dst << src.rdbuf();
    return 0;
}
```

# Kopieren von Dateien

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>
int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream src(arguments[1]);
    std::ofstream dst(arguments[2]);
    dst << src.rdbuf();
    return 0;
}
```

- ofstream zum Schreiben in eine Datei
- Erstellt Datei, falls nicht existent

# Kopieren von Dateien

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>
int main(int argc, char** argv)
{
    std::vector<std::string> arguments(argv, argv + argc);
    std::ifstream src(arguments[1]);
    std::ofstream dst(arguments[2]);
    dst << src.rdbuf();
    return 0;
}
```

- ofstream zum Schreiben in eine Datei
- Erstellt Datei, falls nicht existent

- Schreiben in ostream wie gehabt mit <<
- .rdbuf() greift auf internen Buffer des Streams zu
- Schreibt gesamten Inhalt von src nach dst

## Casts

- Transformation von einem Typ in einen anderen über Casts
- Aus C bekannt: C-Style Casts (*(T value)*)
  - In C++ aus Rückwärtskompatibilitätsgründen auch möglich

# Casts

- Transformation von einem Typ in einen anderen über Casts
- Aus C bekannt: C-Style Casts (*(T) value*)
  - In C++ aus Rückwärtskompatibilitätsgründen auch möglich
- C++ enthält vier verschiedene Casts
  - `static_cast`
  - `dynamic_cast`
  - `const_cast`
  - `reinterpret_cast`

## Casts

- Transformation von einem Typ in einen anderen über Casts
- Aus C bekannt: C-Style Casts (*(T) value*)
  - In C++ aus Rückwärtskompatibilitätsgründen auch möglich
- C++ enthält vier verschiedene Casts
  - `static_cast`
  - `dynamic_cast`
  - `const_cast`
  - `reinterpret_cast`
- C-Style Casts verhalten sich je nach Ein- und Ausgabetyt wie `static_cast`, `const_cast` oder `reinterpret_cast`
  - Führt leicht zu unerwünschtem Verhalten
  - Daher nicht empfohlen

# Casts

- Transformation von einem Typ in einen anderen über Casts
- Aus C bekannt: C-Style Casts (*(T) value*)
  - In C++ aus Rückwärtskompatibilitätsgründen auch möglich
- C++ enthält vier verschiedene Casts
  - `static_cast`
  - `dynamic_cast`
  - `const_cast`
  - `reinterpret_cast`
- C-Style Casts verhalten sich je nach Ein- und Ausgabetyt wie `static_cast`, `const_cast` oder `reinterpret_cast`
  - Führt leicht zu unerwünschtem Verhalten
  - Daher nicht empfohlen
- Syntax: `static_cast<T>(expr)`, *T* ist Zieltyp, *expr* zu castender Ausdruck
  - Andere Casts äquivalent

# Anwendungsgebiete der Casts

## ■ `static_cast`

- Klassischer Cast, transformiert einen Wert in einen anderen
- Bsp.: `static_cast<int>(2.5)`
- Deckt die meisten Casts in Quellcode ab
- Typ muss eine (sichere) Transformation erlauben



# Anwendungsgebiete der Casts

## ■ `static_cast`

- Klassischer Cast, transformiert einen Wert in einen anderen
- Bsp.: `static_cast<int>(2.5)`
- Deckt die meisten Casts in Quellcode ab
- Typ muss eine (sichere) Transformation erlauben

## ■ `const_cast`

- Entfernt Constness
- Bsp.: `const char x; const_cast<char*>(x)`
- Gefährlich: Modifikation von **ursprünglich** als `const` deklarierten Objekten ist undefiniertes Verhalten
- Daher nur in Ausnahmefällen sinnvoll: Verwendung von APIs, die nicht const-correct sind
- Generell vermeiden, Verwendung begründend kommentieren

# Anwendungsgebiete der Casts

## ■ `static_cast`

- Klassischer Cast, transformiert einen Wert in einen anderen
- Bsp.: `static_cast<int>(2.5)`
- Deckt die meisten Casts in Quellcode ab
- Typ muss eine (sichere) Transformation erlauben

## ■ `const_cast`

- Entfernt Constness
- Bsp.: `const char x; const_cast<char*>(x)`
- Gefährlich: Modifikation von **ursprünglich** als `const` deklarierten Objekten ist undefiniertes Verhalten
- Daher nur in Ausnahmefällen sinnvoll: Verwendung von APIs, die nicht const-correct sind
- Generell vermeiden, Verwendung begründend kommentieren

## ■ `reinterpret_cast` und `dynamic_cast`

- Selten notwendig, daher hier (noch) nicht behandelt

# Type-Alias

- `using` definiert alternative Namen für Typen
- Syntax: `using new-type = type;`

# Type-Alias

- `using` definiert alternative Namen für Typen
- Syntax: `using new-type = type;`
- Nützlich, wenn der Typ nachträglich änderbar sein soll
  - Bsp: `using real = float;`
  - Ist `float` zu ungenau, kann durch Änderung von nur einer Zeile auf `double` umgestellt werden

# Type-Alias

- `using` definiert alternative Namen für Typen
- Syntax: `using new-type = type;`
- Nützlich, wenn der Typ nachträglich änderbar sein soll
  - Bsp: `using real = float;`
  - Ist `float` zu ungenau, kann durch Änderung von nur einer Zeile auf `double` umgestellt werden
- Abkürzen langer Typnamen
  - Bsp: `using ItemIterator = std::vector<std::tuple<std::string, int>>::iterator;`

# Type-Alias

- `using` definiert alternative Namen für Typen
- Syntax: `using new-type = type;`
- Nützlich, wenn der Typ nachträglich änderbar sein soll
  - Bsp: `using real = float;`
  - Ist `float` zu ungenau, kann durch Änderung von nur einer Zeile auf `double` umgestellt werden
- Abkürzen langer Typnamen
  - Bsp: `using ItemIterator = std::vector<std::tuple<std::string, int>>::iterator;`
- Sichtbarkeit unterliegt den selben Scopingregeln wie für Variablen

## Type-Alias

- `using` definiert alternative Namen für Typen
- Syntax: `using new-type = type;`
- Nützlich, wenn der Typ nachträglich änderbar sein soll
  - Bsp: `using real = float;`
  - Ist `float` zu ungenau, kann durch Änderung von nur einer Zeile auf `double` umgestellt werden
- Abkürzen langer Typnamen
  - Bsp: `using ItemIterator = std::vector<std::tuple<std::string, int>>::iterator;`
- Sichtbarkeit unterliegt den selben Scopingregeln wie für Variablen
- Neu in C++11 (früher: `typedef`, ohne Unterstützung von Templates)

## auto

- Typ-Deduzierung ist populäres Feature vieler moderner Sprachen
- Seit C++11 auch in C++



# auto

- Typ-Deduzierung ist populäres Feature vieler moderner Sprachen
- Seit C++11 auch in C++
- **auto** ist Platzhalter für Typ einer Variable
- Verwendung: **auto** *var* = *expr*;
  - Typ von *var* entspricht dem Grundtyp von *expr*

## auto

- Typ-Deduzierung ist populäres Feature vieler moderner Sprachen
- Seit C++11 auch in C++
- `auto` ist Platzhalter für Typ einer Variable
- Verwendung: `auto var = expr;`
  - Typ von `var` entspricht dem Grundtyp von `expr`
- Für den Grundtyp eines Typs werden Referenzmodifikatoren ignoriert
  - Grundtyp von `int&` ist `int`

## auto

- Typ-Deduzierung ist populäres Feature vieler moderner Sprachen
- Seit C++11 auch in C++
- `auto` ist Platzhalter für Typ einer Variable
- Verwendung: `auto var = expr;`
  - Typ von `var` entspricht dem Grundtyp von `expr`
- Für den Grundtyp eines Typs werden Referenzmodifikatoren ignoriert
  - Grundtyp von `int&` ist `int`
- `const` wird ignoriert
  - Nur außenliegendes `const` (später mehr)

## auto II

```
const double& foo();  
auto bam = foo();
```

- Welchen Typ hat bam?

## auto II

```
const double& foo();  
auto bam = foo();
```

- Welchen Typ hat bam?
  - Antwort: `double`

## auto II

```
const double& foo();  
auto bam = foo();
```

- Welchen Typ hat bam?
  - Antwort: `double`
- Const- und Referenzmodifikatoren wie für normale Typen
  - `const auto& bam = foo();`

## auto II

```
const double& foo();  
auto bam = foo();
```

- Welchen Typ hat bam?
  - Antwort: `double`
- Const- und Referenzmodifikatoren wie für normale Typen
  - `const auto& bam = foo();`
- Nur Variablen, keine `auto`-Funktionsparameter

# Structured Bindings (C++17)

## ■ Beispiel

```
std::tuple<int, int> bar(10, 5);
auto [a, b] = bar;
std::cout << a << << ", " << b << "\n";
```



# Structured Bindings (C++17)

## ■ Beispiel

```
std::tuple<int, int> bar(10, 5);  
auto [a, b] = bar;  
std::cout << a << << ", " << b << "\n";
```

## ■ Initialisierung mehrerer Variablen in einer Anweisung

## Structured Bindings (C++17)

- Beispiel

```
std::tuple<int, int> bar(10, 5);  
auto [a, b] = bar;  
std::cout << a << << ", " << b << "\n";
```

- Initialisierung mehrerer Variablen in einer Anweisung
- Verwendbar u.a. für `std::tuple`, Arrays, Structs

# Structured Bindings (C++17)

## ■ Beispiel

```
std::tuple<int, int> bar(10, 5);
auto [a, b] = bar;
std::cout << a << << ", " << b << "\n";
```

## ■ Initialisierung mehrerer Variablen in einer Anweisung

## ■ Verwendbar u.a. für std::tuple, Arrays, Structs

## ■ Beispiel: Iterieren über eine std::map

```
std::map<int, std::string> myMap{{3, "foo"}, {2, "bar"}};
for (const auto& [k,v] : myMap)
    std::cout << k << " " << v << std::endl;
```



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

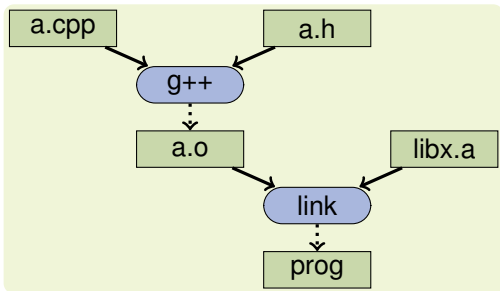
# Die Toolchain Teil 2



# Übersicht

- Buildsystem: make

## Buildsystem: Problematik



- Zwischen Dateien bestehen Abhängigkeiten.
- Was muss alles in welcher Reihenfolge neu erzeugt werden, wenn sich eine Datei ändert?

# Buildsystem: make

## ■ Input

### ■ Makefile

- Liste von Abhängigkeiten (i.d.R. manuell eingegeben)
- Kommandos zum Erzeugen/Aktualisieren abhängiger Dateien

### ■ Zu jeder Datei: Änderungszeitpunkt

## ■ Aktionen

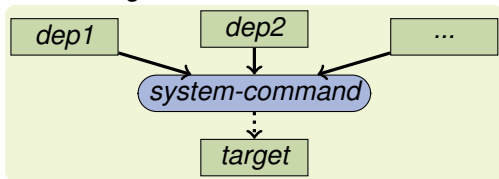
- make erkennt anhand des Dateialters, welche Dateien neu erzeugt werden müssen.
- Ruft entsprechende Kommandos auf

## ■ make *target*

- Erzeugt/Erneuert *target*
- make-Aufruf ohne Target verwendet das erste Target im Makefile
- Verwendet Datei „Makefile“ oder „makefile“, falls vorhanden

# Abhängigkeitsregel im Makefile: Allgemein

## Darstellung



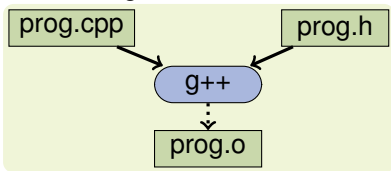
## Im Makefile

```
target: dependencies  
system-command
```



# Abhängigkeitsregel im Makefile: Beispiel

## ■ Darstellung



## ■ Im Makefile

```
prog.o: prog.cpp prog.h  
    g++ prog.cpp -o prog.o
```

## Weiteres zu make

- Variablen

```
OBJS      := pre.o comp.o
ALL_OBJS := $(OBJS) link.o
```

- Automatische Variablen

- \$@: Target einer Regel
- \$<: Erste Dependency einer Regel

- Automatische Regeln

```
.cpp.o:
    g++ -c -o $@ $<
```

- Problematisch

- Automatische Erkennung, welche Header inkludiert werden.

- Weitere Details: s. Literatur

- Im Programmierpraktikum wird Makefile gegeben

# Ein einfaches make-Beispiel

```
1: OBJS      := pre.o comp.o
2: ALL_OBJS := $(OBJS) link.o
3: all:      prog
4: prog:     $(ALL_OBJS)
5:          g++ -o $$ $(ALL_OBJS)
6: .cpp.o:
7:          g++ -c -o $$ $<
8: clean:
9:          rm -f prog $(ALL_OBJS)
```

- Zeile 1: Definiert Variable OBJS

## Ein einfaches make-Beispiel

```
1: OBJS      := pre.o comp.o
2: ALL_OBJS := $(OBJS) link.o
3: all:      prog
4: prog:     $(ALL_OBJS)
5:          g++ -o $$ $(ALL_OBJS)
6: .cpp.o:
7:          g++ -c -o $$ $<
8: clean:
9:          rm -f prog $(ALL_OBJS)
```

- Zeile 1: Definiert Variable OBJS
- Zeile 2: Verwendet Variable

## Ein einfaches make-Beispiel

```
1: OBJS      := pre.o comp.o
2: ALL_OBJS := $(OBJS) link.o
3: all: prog
4: prog: $(ALL_OBJS)
5:      g++ -o $$ $(ALL_OBJS)
6: .cpp.o:
7:      g++ -c -o $$ $<
8: clean:
9:      rm -f prog $(ALL_OBJS)
```

- Zeile 1: Definiert Variable OBJS
- Zeile 2: Verwendet Variable
- Zeile 3: Übliches Pseudo-Target all

## Ein einfaches make-Beispiel

```
1: OBJS      := pre.o comp.o
2: ALL_OBJS := $(OBJS) link.o
3: all:      prog
4: prog: $(ALL_OBJS)
5:          g++ -o $@ $(ALL_OBJS)
6: .cpp.o:
7:          g++ -c -o $@ $<
8: clean:
9:          rm -f prog $(ALL_OBJS)
```

- Zeile 1: Definiert Variable OBJS
- Zeile 2: Verwendet Variable
- Zeile 3: Übliches Pseudo-Target all
- Zeile 4, 5: Regel, wie das Programm prog gelinkt wird.

## Ein einfaches make-Beispiel

```
1: OBJS      := pre.o comp.o
2: ALL_OBJS := $(OBJS) link.o
3: all: prog
4: prog: $(ALL_OBJS)
5:      g++ -o $$ $(ALL_OBJS)
6: .cpp.o:
7:      g++ -c -o $$ $<
8: clean:
9:      rm -f prog $(ALL_OBJS)
```

- Zeile 1: Definiert Variable OBJS
- Zeile 2: Verwendet Variable
- Zeile 3: Übliches Pseudo-Target all
- Zeile 4, 5: Regel, wie das Programm prog gelinkt wird.
- Zeile 6, 7: Automatische Regel: Wie aus einer .cpp- eine .o-Datei wird.

## Ein einfaches make-Beispiel

```
1: OBJS      := pre.o comp.o
2: ALL_OBJS := $(OBJS) link.o
3: all:      prog
4: prog:     $(ALL_OBJS)
5:          g++ -o $$ $(ALL_OBJS)
6: .cpp.o:
7:          g++ -c -o $$ $<
8: clean:
9:          rm -f prog $(ALL_OBJS)
```

- Zeile 1: Definiert Variable OBJS
- Zeile 2: Verwendet Variable
- Zeile 3: Übliches Pseudo-Target all
- Zeile 4, 5: Regel, wie das Programm prog gelinkt wird.
- Zeile 6, 7: Automatische Regel: Wie aus einer .cpp- eine .o-Datei wird.
- Zeile 8, 9: Übliches Pseudo-Target clean: Löscht alle erzeugten Dateien.



## Andere Buildsysteme

- cmake
- Autotools, GNU Build System
- Ant, Maven  
(Java-basiert)
- SCons  
(Python-basiert)



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

# C++-Fehlermeldungen

# Probleme mit Fehlermeldungen

- Fehler sind in der Programmierung fast unvermeidlich
  - In kompilierten Sprachen erkennt der Compiler bereits viele Programmierfehler
  - Mit entsprechendem Warninglevel werden neben Fehlern auch schlechte Codingpraktiken erkannt

# Probleme mit Fehlermeldungen

- Fehler sind in der Programmierung fast unvermeidlich
  - In kompilierten Sprachen erkennt der Compiler bereits viele Programmierfehler
  - Mit entsprechendem Warninglevel werden neben Fehlern auch schlechte Codingpraktiken erkannt
- C++ neigt zu kryptischen Fehlermeldungen
  - Einer der Gründe, die den Einstieg in C++ erschweren
  - In den letzten Jahren wird massiv an der Verbesserung der Fehlermeldungen gearbeitet
  - Bisher mit begrenztem Erfolg

# Probleme mit Fehlermeldungen

- Fehler sind in der Programmierung fast unvermeidlich
  - In kompilierten Sprachen erkennt der Compiler bereits viele Programmierfehler
  - Mit entsprechendem Warninglevel werden neben Fehlern auch schlechte Codingpraktiken erkannt
- C++ neigt zu kryptischen Fehlermeldungen
  - Einer der Gründe, die den Einstieg in C++ erschweren
  - In den letzten Jahren wird massiv an der Verbesserung der Fehlermeldungen gearbeitet
  - Bisher mit begrenztem Erfolg
- Deuten der Fehlermeldungen essentielle Fähigkeit für die Entwicklung in C++

# Probleme mit Streams

```
#include <string>
#include <iostream>

int main()
{
    std::string foo = "bar\n";
    std::cout >> foo;

    return 0;
}
```

# Probleme mit Streams

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'std::__cxx11::string {aka
std::__cxx11::basic_string<char>}')
    std::cout >> foo;
                ^
In file included from /usr/include/c++/6.3.1/iostream:40:0,
    from main.cpp:2:
/usr/include/c++/6.3.1/istream:924:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&&, _Tp&)
operator>>(basic_istream<_CharT, _Traits>&& __is, _Tp& __x)
    ^
/usr/include/c++/6.3.1/istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
    std::cout >> foo;
                ^
In file included from /usr/include/c++/6.3.1/iostream:40:0,
    from main.cpp:2:
/usr/include/c++/6.3.1/istream:808:5: note: candidate: template<class _Traits> std::basic_istream<char, _Traits>&
std::operator>>(std::basic_istream<char, _Traits>&, signed char*)
operator>>(basic_istream<char, _Traits>& __in, signed char* __s)
    ^
/usr/include/c++/6.3.1/istream:808:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<char, _Traits>'
    std::cout >> foo;
                ^
In file included from /usr/include/c++/6.3.1/iostream:40:0,
    from main.cpp:2:
/usr/include/c++/6.3.1/istream:803:5: note: candidate: template<class _Traits> std::basic_istream<char, _Traits>&
std::operator>>(std::basic_istream<char, _Traits>&, unsigned char*)
operator>>(basic_istream<char, _Traits>& __in, unsigned char* __s)
    ^
/usr/include/c++/6.3.1/istream:803:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<char, _Traits>'
    std::cout >> foo;
                ^
In file included from /usr/include/c++/6.3.1/iostream:40:0,
    from main.cpp:2:
```

# Probleme mit Streams

```
main.cpp: In function 'int main()':
main.cpp:7:18: error: no match for 'operator+' (operand types are 'std::ostream (aka std::basic_ostream<char>)' and 'std::__cxx11::string (aka std::__cxx11::basic_string<char>)')
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:94:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_istream<_CharT, _Traits> &operator<>(std::basic_istream<_CharT, _Traits>, _Tp)
    operator<>(basic_istream<_CharT, _Traits>& __in, _Tp __t) = 0
    ^
/usr/include/c++/6.3.1/iostream:94:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<_CharT, _Traits>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:80:5: note: candidate: template<class _Traits> std::basic_istream<char, _Traits> &operator<>(std::basic_istream<char, _Traits>, signed char)
    operator<>(basic_istream<char, _Traits>& __in, signed char __c) = 0
    ^
/usr/include/c++/6.3.1/iostream:80:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<char, _Traits>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:80:5: note: candidate: template<class _Traits> std::basic_istream<char, _Traits> &operator<>(std::basic_istream<char, _Traits>, unsigned char)
    operator<>(basic_istream<char, _Traits>& __in, unsigned char __c) = 0
    ^
/usr/include/c++/6.3.1/iostream:80:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<char, _Traits>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:76:5: note: candidate: template<class _Traits> std::basic_istream<char, _Traits> &operator<>(std::basic_istream<char, _Traits>, signed char)
    operator<>(basic_istream<char, _Traits>& __in, signed char __c) = 0
    ^
/usr/include/c++/6.3.1/iostream:76:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<char, _Traits>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:75:5: note: candidate: template<class _Traits> std::basic_istream<char, _Traits> &operator<>(std::basic_istream<char, _Traits>, unsigned char)
    operator<>(basic_istream<char, _Traits>& __in, unsigned char __c) = 0
    ^
/usr/include/c++/6.3.1/iostream:75:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<char, _Traits>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:93:0: note: candidate: template<class _CharT, class _Traits> std::basic_istream<_CharT, _Traits> &operator<>(std::basic_istream<_CharT, _Traits>, _CharT)
    operator<>(basic_istream<_CharT, _Traits>& __in, _CharT __c) = 0
    ^
/usr/include/c++/6.3.1/iostream:93:0: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<_CharT, _Traits>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/iostream:62:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/iostream:94:0: note: candidate: template<class _CharT2, class _Traits2> std::basic_istream<_CharT2, _Traits2> &operator<>(std::basic_istream<_CharT2, _Traits2>, _CharT2)
    operator<>(basic_istream<_CharT2, _Traits2>& __in, _CharT2 __c) = 0
    ^
/usr/include/c++/6.3.1/iostream:94:0: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<_CharT2, _Traits2>'
    std::cout << flog
    ^
In file included from /usr/include/c++/6.3.1/string:53:0,
                 from main.cpp:1:
/usr/include/c++/6.3.1/string:1461:5: note: candidate: template<class _CharT, class _Traits, class _Alloc> std::basic_istream<_CharT, _Traits> &operator<>(std::basic_istream<_CharT, _Traits>, std::__cxx11::basic_string<_CharT, _Traits, _Alloc>)
    operator<>(basic_istream<_CharT, _Traits>& __in,
    ^
/usr/include/c++/6.3.1/string:1461:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_istream<_CharT, _Traits>'
    std::cout << flog
    ^
```



# Probleme mit Streams

```
main.cpp: In function 'int main()':
main.cpp:7:18: error: no match for 'operator' (operand types are 'std::ostream (aka std::basic_ostream<char>)' and 'std::__cxx11::string (aka std::__cxx11::basic_string<char>)')
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:94:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_ostream<_CharT, _Traits> std::operator<>(std::basic_ostream<_CharT, _Traits>, const _Tp&) const
operator<>(std::basic_ostream<_CharT, _Traits>& __os, const _Tp& __t)
               ^
/usr/include/c++/6.3.1/ostream:94:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<_CharT, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:80:5: note: candidate: template<class _Traits> std::basic_ostream<char, _Traits> std::operator<>(std::basic_ostream<char, _Traits>, signed char) const
operator<>(std::basic_ostream<char, _Traits>& __os, signed char __c)
               ^
/usr/include/c++/6.3.1/ostream:80:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<char, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:80:5: note: candidate: template<class _Traits> std::basic_ostream<char, _Traits> std::operator<>(std::basic_ostream<char, _Traits>, unsigned char) const
operator<>(std::basic_ostream<char, _Traits>& __os, unsigned char __c)
               ^
/usr/include/c++/6.3.1/ostream:80:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<char, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:76:5: note: candidate: template<class _Traits> std::basic_ostream<char, _Traits> std::operator<>(std::basic_ostream<char, _Traits>, int) const
operator<>(std::basic_ostream<char, _Traits>& __os, int __i)
               ^
/usr/include/c++/6.3.1/ostream:76:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<char, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:76:5: note: candidate: template<class _Traits> std::basic_ostream<char, _Traits> std::operator<>(std::basic_ostream<char, _Traits>, long) const
operator<>(std::basic_ostream<char, _Traits>& __os, long __l)
               ^
/usr/include/c++/6.3.1/ostream:76:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<char, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:76:5: note: candidate: template<class _Traits> std::basic_ostream<char, _Traits> std::operator<>(std::basic_ostream<char, _Traits>, long long) const
operator<>(std::basic_ostream<char, _Traits>& __os, long long __ll)
               ^
/usr/include/c++/6.3.1/ostream:76:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<char, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/ostream:934:0:
/usr/include/c++/6.3.1/bits/ostream.tcc:923:5: note: candidate: template<class _CharT, class _Traits> std::basic_ostream<_CharT, _Traits> std::operator<>(std::basic_ostream<_CharT, _Traits>, const _CharT&) const
operator<>(std::basic_ostream<_CharT, _Traits>& __os, const _CharT& __c)
               ^
/usr/include/c++/6.3.1/bits/ostream.tcc:923:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<_CharT, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/ostream:60:0,
                 from main.cpp:2:
/usr/include/c++/6.3.1/bits/ostream.tcc:955:5: note: candidate: template<class _CharT2, class _Traits> std::basic_ostream<_CharT2, _Traits> std::operator<>(std::basic_ostream<_CharT, _Traits>, const _CharT2&) const
operator<>(std::basic_ostream<_CharT2, _Traits>& __os, const _CharT2& __c)
               ^
/usr/include/c++/6.3.1/bits/ostream.tcc:955:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_ostream<_CharT2, _Traits>'
    std::cout << flog;
                  ^
In file included from /usr/include/c++/6.3.1/string:53:0,
                 from main.cpp:1:
/usr/include/c++/6.3.1/bits/basic_string.tcc:1441:5: note: candidate: template<class _CharT, class _Traits, class _Alloc> std::basic_string<_CharT, _Traits> std::operator<>(std::basic_string<_CharT, _Traits>, const _Alloc&) const
operator<>(std::basic_string<_CharT, _Traits>& __s, const _Alloc& __a)
               ^
/usr/include/c++/6.3.1/bits/basic_string.tcc:1441:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream (aka std::basic_ostream<char>)' is not derived from 'std::basic_string<_CharT, _Traits>'
    std::cout << flog;
                  ^
```

# Unwichtige Informationen ausblenden

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'std::__cxx11::string {aka
std::__cxx11::basic_string<char>}')
std::cout >> foo;
      ^
In file included from /usr/include/c++/6.3.1/iostream:40:0,
from main.cpp:2:
/usr/include/c++/6.3.1/istream:924:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&&, _Tp&)
operator>>(basic_istream<_CharT, _Traits>&&, __is, _Tp& __x)
      ^
/usr/include/c++/6.3.1/istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
std::cout >> foo;
      ^
...
In file included from /usr/include/c++/6.3.1/string:53:0,
from main.cpp:1:
/usr/include/c++/6.3.1/bits/basic_string.tcc:1441:5: note: candidate: template<class _CharT, class _Traits, class _Alloc> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&, std::__cxx11::basic_string<_CharT, _Traits, _Alloc>&)
operator>>(basic_istream<_CharT, _Traits>&, __in,
      ^
/usr/include/c++/6.3.1/bits/basic_string.tcc:1441:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
std::cout >> foo;
      ^
```

# Unwichtige Informationen ausblenden

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'std::__cxx11::string {aka
std::__cxx11::basic_string<char>}')
std::cout >> foo;
^
In file included from /usr/include/c++/6.3.1/iostream:40:0,
from main.cpp:2:
/usr/include/c++/6.3.1/istream:924:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&&, _Tp&)
operator>>(basic_istream<_CharT, _Traits>&& __is, _Tp& __x)
^
/usr/include/c++/6.3.1/istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
std::cout >> foo;
^
...
In file included from /usr/include/c++/6.3.1/string:53:0,
from main.cpp:1:
/usr/include/c++/6.3.1/bits/basic_string.tcc:1441:5: note: candidate: template<class _CharT, class _Traits, class _Alloc> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&, std::__cxx11::basic_string<_CharT, _Traits, _Alloc>&)
operator>>(basic_istream<_CharT, _Traits>& __in,
^
/usr/include/c++/6.3.1/bits/basic_string.tcc:1441:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
std::cout >> foo;
^
```

- Pfade zur Standardbibliothek sind uninteressant

# Unwichtige Informationen ausblenden

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'std::__cxx11::string {aka
std::__cxx11::basic_string<char>}')
std::cout >> foo;
               ^
istream:924:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&&, _Tp&)
operator>>(basic_istream<_CharT, _Traits>&& __is, _Tp& __x)
               ^
istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
std::cout >> foo;
               ^
...
```

## Unwichtige Informationen ausblenden

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'std::__cxx11::string {aka
std::__cxx11::basic_string<char>}')
std::cout >> foo;
               ^
istream:924:5: note: candidate: template<class _CharT, class _Traits, class _Tp> std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&&, _Tp&)
operator>>(basic_istream<_CharT, _Traits>&& __is, _Tp& __x)
               ^
istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream {aka std::basic_ostream<char>}' is not derived from 'std::basic_istream<_CharT, _Traits>'
std::cout >> foo;
               ^
...
```

- `std::basic_`*something*`<T>` sind Templateklassen mit Typedefs für typische Verwendungen
- `std::basic_istream<char>` ist Typ von `std::istream`
  - Eigentlich: `std::basic_istream<char, std::char_traits<char>>`
- Gilt auch für `std::string` (`std::basic_string<char>`)
- `std::__cxx11` ist interner Namespace für Klassen, die in C++11 anders implementiert sind als in C++03
  - Wird transparent in `std` eingebunden, wenn in C++11 kompiliert wird

## Unwichtige Informationen ausblenden

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream' and 'std::string')
std::cout >> foo;
               ^
istream:924:5: note: candidate: template<class _Tp> std::istream& std::operator>>(std::istream&&, _Tp&)
operator>>(istream&& __is, _Tp& __x)
               ^
istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream' is not derived from 'std::istream'
std::cout >> foo;
               ^
...
basic_string.tcc:1441:5: note: candidate: std::istream& std::operator>>(std::istream&, std::string&)
operator>>(istream& __in, string& __s)
               ^
basic_string.tcc:1441:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream' is not derived from 'std::istream'
std::cout >> foo;
               ^
```

## Unwichtige Informationen ausblenden

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream' and 'std::string')
  std::cout >> foo;
               ^
istream:924:5: note: candidate: template<class _Tp> std::istream& std::operator>>(std::istream&&, _Tp&)
  operator>>(istream&& __is, _Tp& __x)
               ^
istream:924:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream' is not derived from 'std::istream'
  std::cout >> foo;
               ^
...
basic_string.tcc:1441:5: note: candidate: std::istream& std::operator>>(std::istream&, std::string&)
  operator>>(istream& __in, string& __s)
               ^
basic_string.tcc:1441:5: note: template argument deduction/substitution failed:
main.cpp:7:18: note: 'std::ostream' is not derived from 'std::istream'
  std::cout >> foo;
               ^
```

- Fehler werden durch error gekennzeichnet
- note gibt lediglich zusätzliche Informationen über den Fehler
  - Hilfreich, wenn der Fehler im eigenen Code nicht aussagekräftig ist

## Der eigentliche Fehler

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no match for 'operator>>' (operand types are 'std::ostream' and
               'std::string')
    std::cout >> foo;
               ^
```

- In `std::ostream` wird mit `<<` geschrieben, `>>` ist der falsche Operand