

Kapitel 18

Einführung in C++

I find languages that support just one
programming paradigm constraining.

Bjarne Stroustrup

Schlüsselwörter in C

Für ANSI-C sind insgesamt die folgenden 32 Schlüsselwörter definiert. Einen Großteil davon haben wir bereits kennengelernt und verwendet.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Neue Schlüsselwörter in C++

Zusätzlich zu den bereits aus C bekannten Schlüsselwörtern sind in C++ weitere Schlüsselwörter hinzugekommen. Wir werden einen Großteil dieser Schlüsselwörter im weiteren nutzen. Wie die alten Schlüsselwörter können sie nicht für Namen verwendet werden, beispielsweise für Variablen.

asm	dynamic_cast	namespace	reinterpret_cast	try
bool	explicit	new	static_cast	typeid
catch	false	operator	template	typename
class	friend	private	this	using
const_cast	inline	public	throw	virtual
delete	mutable	protected	true	wchar_t

Neue Operatoren in C++

In C++ werden einige ganz neue Operatoren eingeführt. Den Operator für den Globalzugriff verwenden wir schon in diesem Kapitel, die anderen Operatoren kommen erst mit der objektorientierten Programmierung zum Einsatz.

Operator	Bezeichnung
::	Globalzugriff
::	Class-Member Zugriff
.*	Pointer-to-Member Zugriff (direkt)
->*	Pointer-to-Member Zugriff (indirekt)
new	Objekt Allokator
delete	Objekt Deallokator

Gesamtübersicht der Operatoren in C++ (1/4)

Neu in C++

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
:: ::	::var cl::mem	Globalzugriff Class-Member-Zugriff	Zugriffsoperator		17
()	f (x, y)	Funktionsaufruf	Auswertungsoperator	L	16
[]	a [i]	Array-Zugriff	Zugriffsoperator		
->	p->x	Indirekt-Zugriff			
.	a.x	Struktur-Zugriff			
++	x++	Post-Inkrement	Zuweisungsoperator		
--	x--	Post-Dekrement			
!	!x	Logische Verneinung	Logischer Operator	R	15
~	~x	Bitweises Komplement			
++	++x	Pre-Inkrement	Zuweisungsoperator		
--	--x	Pre-Dekrement			
+	+x	Plus x	Arithmetischer Operator		
-	-x	Minus x			

Gesamtübersicht der Operatoren in C++ (2/4)

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
*	*p	Dereferenzierung	Zugriffsoperator	R	15
&	&x	Adressoperator			
()	(type)	Typ-Konvertierung			
sizeof	sizeof (x)	Typ-Speichergröße	Datentyp-Operator		
new	new class	Objekt allokieren			
delete	delete a	Objekt deallokieren			
.*		Pointer-to-Member-Zugriff	Zugriffsoperator	L	14
->*		Pointer-to-Member-Zugriff			
*	x*y	Multiplikation	Arithmetischer Operator	L	13
/	x/y	Division			
%	x%y	Rest bei Division			
+	x+y	Addition	Arithmetischer Operator	L	12
-	x-y	Subtraktion			

Neu in C++

Neu in C++

Gesamtübersicht der Operatoren in C++ (3/4)

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
<<	<code>x<<y</code>	Bitshift links	Bit-Operator	L	11
>>	<code>x>>y</code>	Bitshift rechts			
<	<code>x<y</code>	Kleiner als	Vergleichsoperator	L	10
<=	<code>x<=y</code>	Kleiner oder gleich			
>	<code>x>y</code>	Größer als			
>=	<code>x>=y</code>	Größer oder gleich			
= =	<code>x==y</code>	Gleich	Vergleichsoperator	L	9
!=	<code>x!=y</code>	Ungleich			
&	<code>x&y</code>	Bitweises und	Bit-Operator	L	8
^	<code>x ^ y</code>	Bitweises entweder oder	Bit-Operator	L	7
	<code>x y</code>	Bitweises oder	Bit-Operator	L	6
&&	<code>x && y</code>	Logisches und	Logischer Operator	L	5
	<code>x y</code>	Logisches oder	Logischer Operator	L	4

Gesamtübersicht der Operatoren in C++ (4/4)

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
? :	$x ? y : z$	Bedingte Auswertung	Auswertungsoperator	L	3
=	$x = y$	Wertzuweisung	Zuweisungsoperator	R	2
+=	$x += y$	Operation mit anschließender Zuweisung			
-=	$x -= y$				
*=	$x * = y$				
/=	$x / = y$				
%=	$x \% = y$				
&?	$x \& = y$				
^=	$x ^ = y$				
=	$x = y$				
<<=	$x << = y$				
>>=	$x >> = y$				
,	x , y	Sequentielle Auswertung	Auswertungsoperator	L	1

Erweiterter Kommentarstil

C++ bietet eine zusätzliche Möglichkeit, Kommentare zu erfassen.

```
y = 2; /* C-Style Kommentar */  
x = 1; // Hier steht ein Kommentar für den Rest der Zeile
```

Bekannter Kommentarstil aus C (weiter gültig)

Kommentar bis zum Ende der Zeile im C++-Stil

Die Möglichkeit, Kommentare auf Folgezeilen zu verlängern, wird beim Lesen von Code leicht übersehen und sollte nicht eingesetzt werden.

```
z = 3; // Ein C++ Kommentar kann (unüblich!) auch so -> \  
z = x * y; fortgesetzt werden
```

Der „Backslash“ verlängert den Kommentar auf die nächste Zeile

Automatische Typisierung von Aufzählungstypen

C++ bietet eine „automatische Typisierung“, die explizite Angabe des Datentyps bei Aufzählungstypen kann damit entfallen.

Verwendung eines enum in C, Datentyp enum muss mit angegeben werden

Bei Verwendung des enum in C++ wird automatisch typisiert

```
enum wochehtag {Mo, Di, Mi, Do, Fr, Sa, So};  
enum wochehtag wt1; /* C Stil */  
wochehtag wt2; // C++ Stil
```

Automatische Typisierung von Strukturen

Die automatische Typisierung gilt auch für Strukturen

```
struct punkt  
{  
    int x;  
    int y;  
};
```

Verwendung der struct in
C

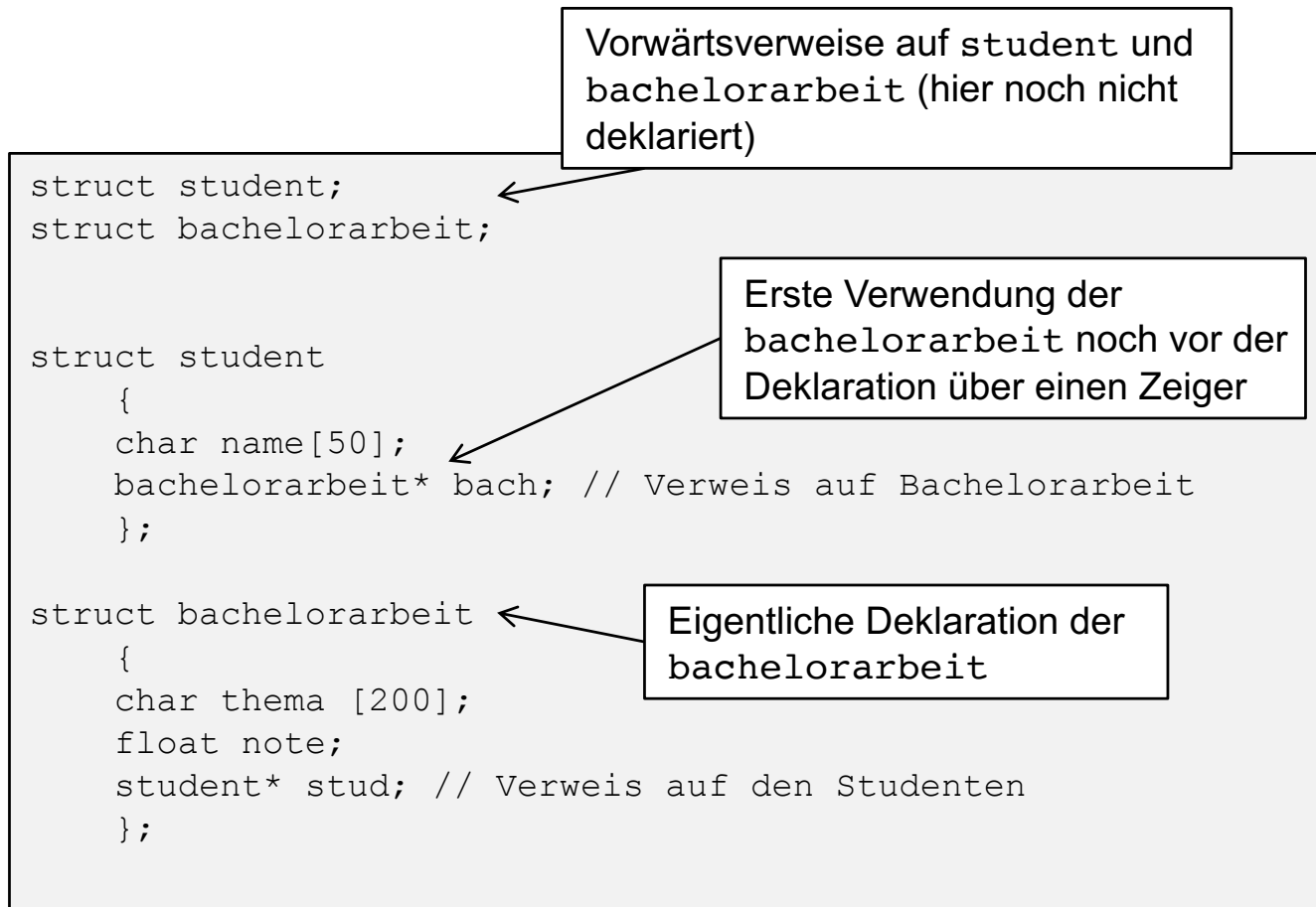
```
struct punkt p1; /* C Stil */
```

```
punkt p2; // C++ Stil
```

Verwendung der struct in C++
wird automatisch typisiert

Vorwärtsverweise

C++ erlaubt Vorwärtsverweise innerhalb von Strukturen, damit kann auf Strukturen verwiesen werden, die an der Stelle des Verweises noch nicht definiert sind. Dies ist insbesondere notwendig, wenn es sich um Zirkelverweise handelt.



Der Datentyp bool

C++ bietet anders als C einen eigenen Datentyp `bool`, um Wahrheitswerte (`true` und `false`) zu speichern.

```
bool b1, b2, b3, b4;

b1 = true;
b2 = !b1;
b3 = 3 > 2;

if( b2 != false)
{
    b4 = b1 || b2;
}
```

Der Datentyp `bool` ist kompatibel mit dem `int` Datentyp, ist aber nicht identisch. Er verhält sich wie ein `int`, der nur die Werte 0 und 1 aufnehmen kann.

```
bool b = 7;

int ausgabe = b;

printf( "Der Wert von ausgabe ist '%d'\n", ausgabe);
```

Der Wert von ausgabe ist '1'

Verwendung von Konstanten

In C++ können (und sollten) Konstanten anstelle symbolischer Konstanten verwendet werden, auch für Informationen die zur Compile-Zeit verfügbar sein müssen.

Die Konstante `anzahl` wird im Quellcode definiert.

```
const int anzahl = 10;  
  
int array[anzahl];
```

Der definierte konstante Wert kann bereits zur Übersetzungszeit verwendet werden

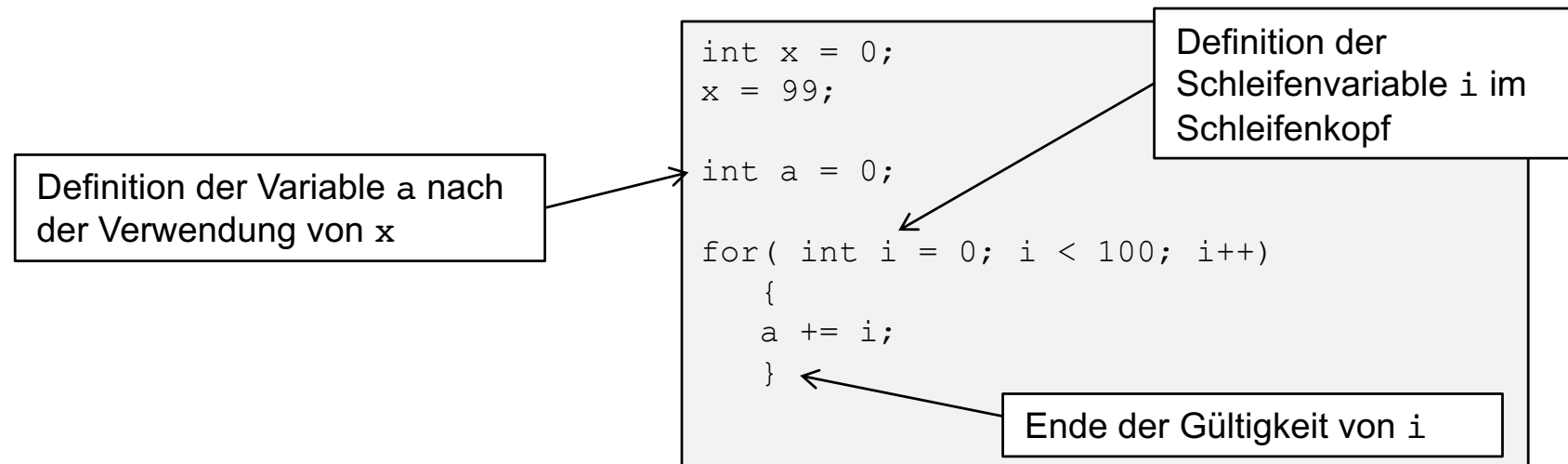
Zur Erinnerung: Dieses Vorgehen war in C nicht möglich, hier musste eine Konstante mit Hilfe des Präprozessors vorgegeben werden. Das oben gezeigte Vorgehen führt in C zu einem Fehler:

```
#define ANZAHL 10  
  
int array[ANZAHL]; /* Vorgehen in ANSI-C */
```

Der Text `ANZAHL` wird durch den Präprozessor vor der Übersetzung durch den Wert 10 ersetzt

Definition von Variablen

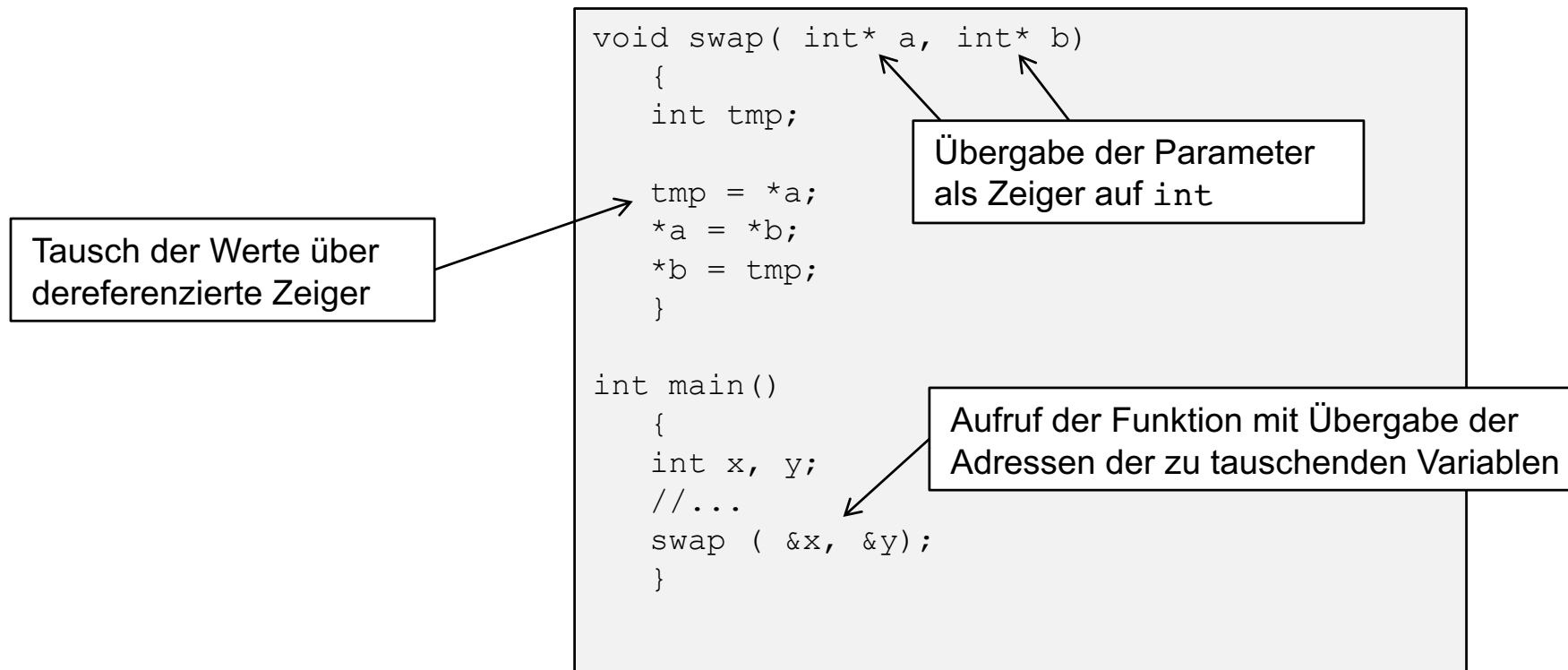
Während in C Variablen am Anfang eines Block definiert werden müssen, können sie in C++ frei eingeführt werden. Voraussetzung ist jeweils, dass sie vor der erstmaligen Benutzung definiert werden.



Verwendung von Referenzen

In C erfolgt die Übergabe von Parametern an eine Funktion immer als Kopie. Wenn die in einer Variable stehenden Werte in einer Funktion geändert werden sollen, dann muss ein Zeiger auf diese Werte übergeben werden. In C++ gibt es mit den sogenannten „Referenzen“ auf Variablen eine elegante und effiziente Alternative.

Wir führen uns die Vorgehensweise in C noch einmal anhand einer „Swap“-Funktion vor Augen.



Verwendung von Referenzen in C++

Eine Referenz ist ein Verweis auf ein bestehendes Element. Die Übergabe per Referenz wird in der Schnittstelle der Funktion durch ein an den Datentyp angestelltes & gekennzeichnet.

Tausch der Werte
über direkte
Zuweisung

```
void swap( int& a, int& b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;
    // ...
    printf( "Vorher; %d %d\n", x, y);

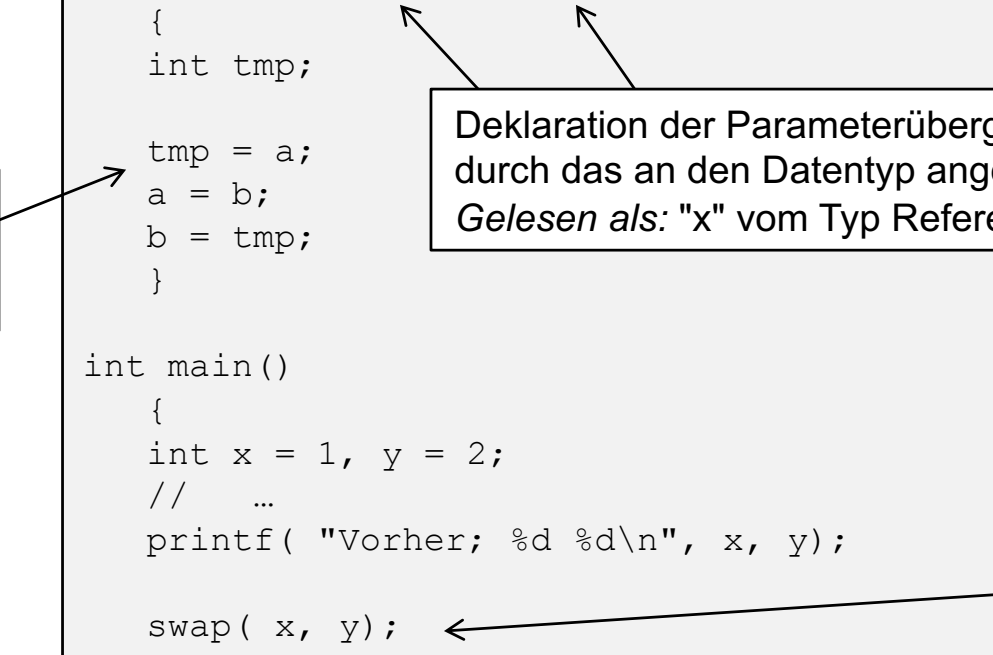
    swap( x, y);

    printf( "Nachher; %d %d\n", x, y);
    return 0;
}
```

Deklaration der Parameterübergabe per Referenz
durch das an den Datentyp angestellte &
Gelesen als: "x" vom Typ Referenz auf Integer

Aufruf der Funktion direkt
mit den Variablen ohne
weitere Dereferenzierung

Vorher: 1 2
Nachher: 2 1



Achtung! Bei der Übergabe eines Wertes per Zeiger sieht man an der Schnittstelle leicht, dass vom Zeiger referenzierten Werte in der Funktion geändert werden könnten.

Bei der Übergabe per Referenz ist dies für den Aufrufer nicht mehr so offensichtlich! Im Grunde handelt es sich bei Referenzen um Zeiger, die bei jeder Übergabe implizit dereferenziert werden.

Referenzen als Rückgabewerte

Referenzen können auch als Rückgabewerte verwendet werden. Ein solcher Rückgabewert einer Funktion kann sogar als L-Value verwendet werden, dass heißt, der der Ausdruck darf auf der linken Seite einer Zuweisung stehen. Wir betrachten dazu eine Alternative der schon bekannten max Funktion:

```
int max1( int a, int b )  
{  
    if( a >= b )  
        return a;  
    else  
        return b;  
}
```

Bekannte max
Funktion

Alternative: Rückgabe
einer Referenz statt
eines Wertes

```
int& max2( int& a, int& b )  
{  
    if( a >= b )  
        return a;  
    else  
        return b;  
}
```

```
void main()  
{  
    int x = 1, y = 2, z;  
  
    z = max1( x, y );  
    printf( "max1: %d\n", z );  
    z = max2 ( x, y );  
    printf( "max2: %d\n", z );  
  
    max2( x, y ) = 4711;  
    printf( "y: %d\n", y );  
}
```

Rückgabe per Wert

Rückgabe per Referenz,
keine Auswirkung

Der Variablen mit dem
größeren Wert, hier y, wird
der Wert 4711 zugewiesen

Einsatz der zurück-
gegebenen Referenz
als L-Value

```
max1: 2  
max2: 2  
y: 4711
```

Einschränkungen von Referenzen

Referenzen sind bei der Übergabe sehr effizient, da nur der Verweis statt des ganzen Elements übergeben wird. Bei einer großen Struktur kann dies ein deutlicher Vorteil sein. Nicht als konstant deklarierte Referenzen können und dürfen in einer Funktion verändert werden. Daher können solchen Referenzen keine konstanten oder konkreten Werte übergeben werden.

```
int& max2( int& a, int& b )  
{  
    if( a >= b )  
        return a;  
    else  
        return b;  
}
```

max Funktion mit Referenzen

```
void main()  
{  
    const int u=3, v=4;  
    int w;  
  
    w = max2( 1, 2 );  
    w = max2( u, v );  
}
```

FEHLER! Übergabe von
Konstanten als Referenz nicht
möglich!

FEHLER! Übergabe
konstanter Variablen als
Referenz nicht möglich!

Konstante Referenzen

Um den Effizienzvorteil von Referenzen zu nutzen und das Risiko der ungewollten Änderung der referenzierten Werte auszuschließen, sollten konstante Referenzen verwendet werden. Diese können auch mit unveränderbaren Werten aufgerufen werden.

Konstanten Referenzen `const int&`

```
int max3( const int& a, const int& b )  
{  
    if( a >= b )  
        return a;  
    else  
        return b;  
}
```

```
void main()  
{  
    int x= 1, y=2;  
    const int u=3, v=4;  
    int w;  
  
    w = max3( 1, 2 );  
    w = max3( u, v );  
    w = max3( x, y );  
}
```

Verwendung mit Konstanten

Verwendung mit konstanten Variablen

Verwendung mit änderbaren Variablen

Vorgegebene Werte in der Funktionsschnittstelle (Default-Werte)

C++ bietet die Möglichkeit, in der Schnittstelle der Funktion „Default-Werte“ anzugeben. Diese Werte werden genutzt, wenn keine anderen Werte angegeben werden.

In der Schnittstellen der Funktion
wird ein Default-Wert definiert

```
void defaultwert( int s = 99)
{
    printf( "Ausgabe s: %d\n",  s);
}
```

```
void main()
{
    defaultwert( 1);
    defaultwert();
}
```

Wenn bei Aufruf der Funktion keine
Werte übergeben werden, wird der
angegebene Default-Wert verwendet.

```
Ausgabe s: 1
Ausgabe s: 99
```

Vorgegebene Werte in der Funktionsdeklaration (weiteres Beispiel)

Default-Parameter können immer nur für die „letzten“ Parameter einer Funktion angegeben werden. Es können beim Aufruf der Funktion nur die Argumente vom Ende her weggelassen werden.

```
int add( int a, int b, int c = 0, int d = 0)
{
    return a + b + c + d;
}

void main()
{
    int a, b, c, d;

    a = add( 1, 2);
    b = add( 1, 2, 3);
    c = add( 1, 2, 3, 4);
    d = add( 1, 2, 0, 0);
}
```

Definition einer Funktion mit mehreren Default-Parametern

Unterschiedliche Aufrufe der Funktion

Hat eine Funktion mit Default-Werten einen Prototypen, gehören die Default-Werte ausschließlich in den Prototypen.

```
extern int add( int a, int b, int c = 0, int d = 0);

int add( int a, int b, int c, int d)
{
    return a + b + c + d;
}
```

Default-Werte im Prototypen sind dort auch ohne den Quellcode der Funktion von außen sichtbar

Inline Funktionen

Insbesondere bei der objektorientierten Programmierung entstehen häufig sehr kleine Funktionen, die einen Aufruf als Funktion „nicht lohnen“. C++ bietet als Lösung „inline“ Funktionen an. Eine inline-Funktion wird, falls möglich, vom Compiler anstelle des Funktionsaufrufes direkt implementiert. In dem Fall muss kein Aufruf der Funktion mit Parameterübergabe erfolgen.

Kennzeichnung der
Funktion als inline

```
inline int max( int a, int b)
{
    return a > b ? a : b;
}
```

```
void main()
{
    int x = 10, y = 100;

    int m = max( x, y );
}
```

Durch inline-Übersetzung entfällt
der Funktionsaufruf und es entsteht
praktisch der folgende Code

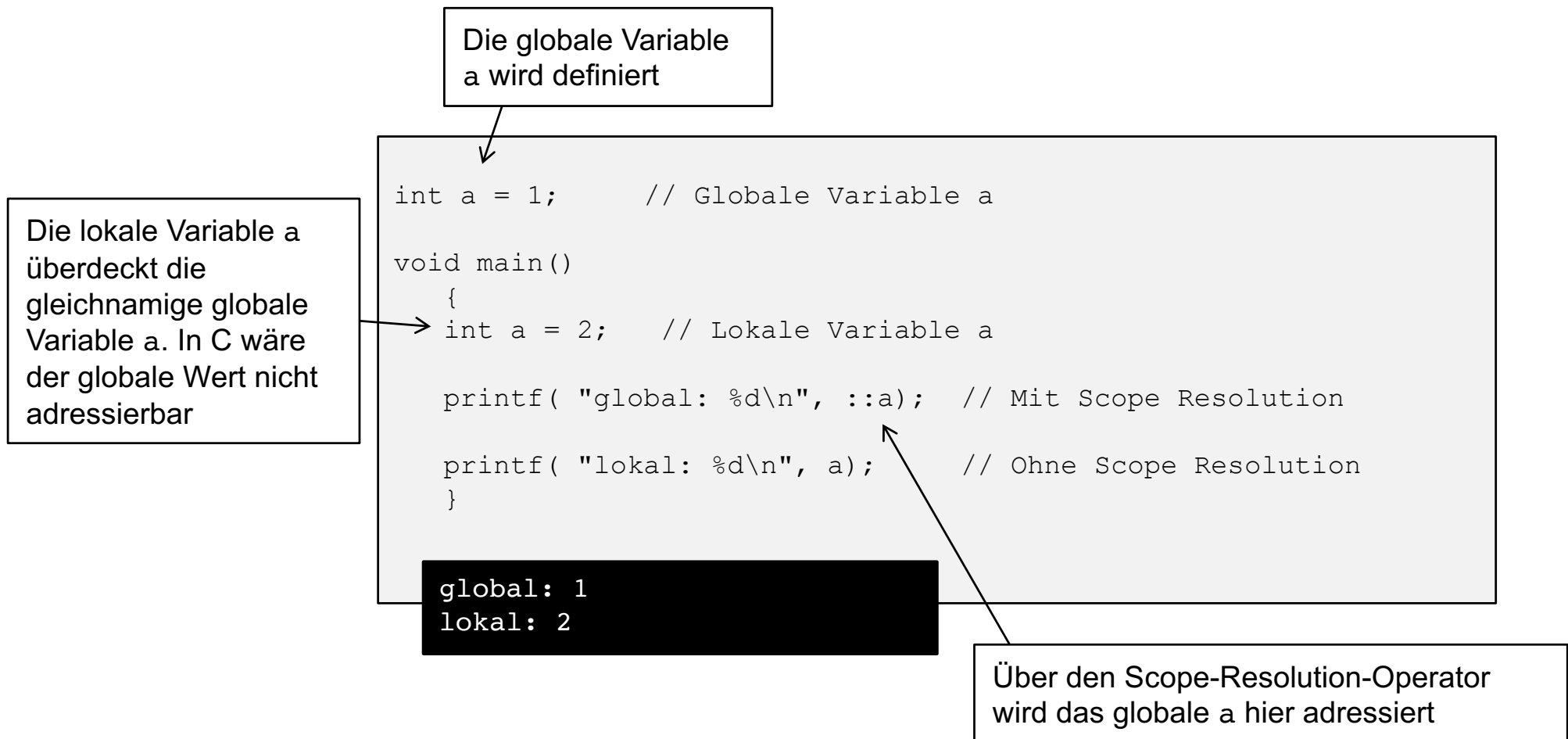
```
void main()
{
    int x = 10, y = 100;

    int m = x > y ? x : y;
}
```

Wird eine inline-Funktion an mehreren Stellen des Programms verwendet, führt dies zu mehr Code. Wie in anderen Fällen, die Sie schon kennen, werden hier mögliche Laufzeitgewinne mit Speicher bezahlt.

Scope Resolution Operator

Anders als in C können „verdeckte“ globale Elemente sichtbar gemacht werden. Hierzu wird der „Scope-Resolution-Operator“ verwendet.



Überladen von Funktionen

Die wichtigste, nicht-objektorientierte Funktionalität von C++ ist das Überladen von Funktionen. Eine typische Situation **ohne** Überladung von Funktionen sieht folgendermaßen aus: Gegeben seien die folgenden Strukturen und deren typische Ausgabe in C.

```
struct punkt
{
    int x;
    int y;
};
```

```
struct vektor
{
    int x;
    int y;
    int z;
};
```

Ausgabefunktionen
mit unterschiedlichen
Namen je Datentyp.
Gleichnamige
Funktionen würden
zum Fehler führen

Aufruf der
unterschiedlichen
Funktionen zur
Ausgabe

```
void print_punkt( punkt p)
{
    printf( "Punkt: (%d, %d)\n", p.x, p.y);
}

void print_vektor( vektor v)
{
    printf( "Vektor: (%d, %d, %d)\n", v.x, v.y, v.z);
}

void main ()
{
    struct punkt p = {5, 4};
    struct vektor v = {1, 2, 3};
    print_punkt( p);
    print_vektor( v);
}
```

```
Punkt (5, 4)
Vektor (1, 2, 3)
```

Überladen von Funktionen

In C++ können Funktionen überladen werden. Das bedeutet, dass Funktionen nicht nur anhand ihres Namens, sondern auch anhand ihrer Parametersignatur unterschieden werden.

Definition je einer
Ausgabefunktion je Datentyp.
Die Namen der Funktionen
unterscheiden sich nicht,
lediglich deren
Parametersignaturen

Aufruf der Funktion
print(punkt)

Aufruf der Funktion
print(vektor)

```
void print( punkt p)
{
    printf( "Punkt: (%d, %d)\n", p.x, p.y);
}

void print( vektor v)
{
    printf( "Vektor: (%d, %d, %d)\n", v.x, v.y, v.z);
}

void main ()
{
    punkt p = {5, 4};
    vektor v = {1, 2, 3};
    print ( p);
    print ( v);
}
```

```
Punkt (5, 4)
Vektor (1, 2, 3)
```

Parametersignatur von Funktionen

Zur Unterscheidung und Auswahl (überladener) Funktionen dient neben dem Funktionsnamen die Parametersignatur. Zu der Signatur gehören Anzahl, Reihenfolge und Typ der übergebenen Parameter. Der Typ des Rückgabewertes geht nicht in die Parametersignatur ein. Ebenso werden Default-Parameter nicht berücksichtigt.

```
int fkt( int a) {return 0;}  
void fkt( int a) {return;}  
  
void main()  
{  
    fkt( 1);  
}
```

Fehler! Die Funktionen sind nicht unterscheidbar, da Name und Parametersignaturen identisch sind

Eine Funktion kann ohne Information zum erwarteten Rückgabetyt aufgerufen werden, daher darf dieser kein Unterscheidungskriterium sein

```
int fkt( int a) {return a;}  
int fkt( int a, int b = 10) {return a + b;}  
  
void main()  
{  
    fkt( 1);  
}
```


Die Funktionen sind im Aufruf nicht unterscheidbar, da Name und Parametersignatur identisch sind

Mehrdeutigkeit! Der Compiler könnte nicht unterscheiden, welche der beiden Funktionen gemeint wäre

Dekorieren von Funktionsnamen in C++

Um das Überladen von Funktionen zu ermöglichen, „dekoriert“ C++ die Funktionsnamen, indem Informationen zu den Parametern an Namen angehängt werden. Die genauen Regeln zur Bildung dieser Namen müssen uns dabei nicht interessieren. Sie sind im C++-Standard definiert. Damit ist sichergestellt, dass alle Compiler die Namen gleich erzeugen und Module die von unterschiedlichen C++-Compilern übersetzt worden sind, auch durch den Linker zusammengeführt werden können.

Der vom Compiler generierte Funktionsname im erzeugten Objektcode lautet `xxx_Ficf` und beinhaltet die Parameter `int`, `char` und `float`



```
void xxx( int a, char b, float c);
```

Überladen von Operatoren

In C++ können auch Operatoren als Funktionen dargestellt werden. Damit ist es naheliegend, dass diese wie andere Funktionen überladen werden können.

Wollen wir beispielsweise den `+`-Operator für eine `punkt` Struktur überladen, verwenden wir eine Funktion die zwei Parameter vom Typ `punkt` erwartet und eine Struktur `punkt` als Rückgabewert liefert:

Funktion mit dem Namen `operator+` zur Überladung des `+`-Operators

Funktion erwartet zwei Parameter des Typs `punkt` und liefert den Typ `punkt` als Ergebnis

```
punkt operator+( punkt p1, punkt p2)
{
    punkt ergebnis;
    ergebnis.x = p1.x + p2.x;
    ergebnis.y = p1.y + p2.y;
    return ergebnis;
}
```

```
void main()
{
    punkt x = {1, 2};
    punkt y = {3, 4};
    punkt u,v ;

    u = x + y;
    print( u);
    v = operator+( y, y);
    print( v);
}
```

Aufruf des `+`-Operators in gewohnter Notation

Nutzung des `+`-Operators als Funktionsaufruf

```
Punkt: (4, 6)
Punkt: (6, 8)
```

Überladen von Operatoren

Wir können beispielsweise auch eine skalare Multiplikation überladen, hier soll der Operator einen `int` Wert als Faktor sowie eine Struktur `punkt` erwarten und eine Struktur `punkt` zurückliefern.

```
punkt operator+( punkt p1, punkt p2)
{
    punkt ergebnis;
    ergebnis.x = p1.x + p2.x;
    ergebnis.y = p1.y + p2.y;
    return ergebnis;
}

punkt operator*( int faktor, punkt p)
{
    punkt ergebnis;
    ergebnis.x = faktor * p.x;
    ergebnis.y = faktor * p.y;
    return ergebnis;
}
```

Funktion `operator*` zur Überladung des `*`-Operators. Die Funktion erwartet je einen Parameter vom Typ `int` und `punkt`

```
void main()
{
    punkt x = { 1, 2 };
    punkt y = { 3, 4 };
    punkt u, v, w;

    u = x + y;
    print ( u );
    v = operator+( y, y );

    print( v );
    w = 2 * v;
    print( w );
    print( operator*( 3, v ) );
}
```

Nutzung des `*`-Operators als Operator

Nutzung des `*`-Operators als Funktionsaufruf

Punkt: (4, 6)
Punkt: (6, 8)
Punkt: (12, 16)
Punkt: (18, 24)

Das Überladen der eingebauten Operatoren (beispielsweise zur Addition von Integerwerten) ist nicht möglich.

Deklaration von C-Funktionen in C++

Wenn wir C-Funktionen in ein C++-Programm einbinden wollen, dann müssen wir dem Compiler mitteilen, dass die Funktionsnamen nicht dekoriert werden sollen. Dazu deklarieren wir die Funktionen als extern "C". Das kann für eine einzelne Funktion oder im Block geschehen.

```
extern "C" void abc(int, char b, float c);
```

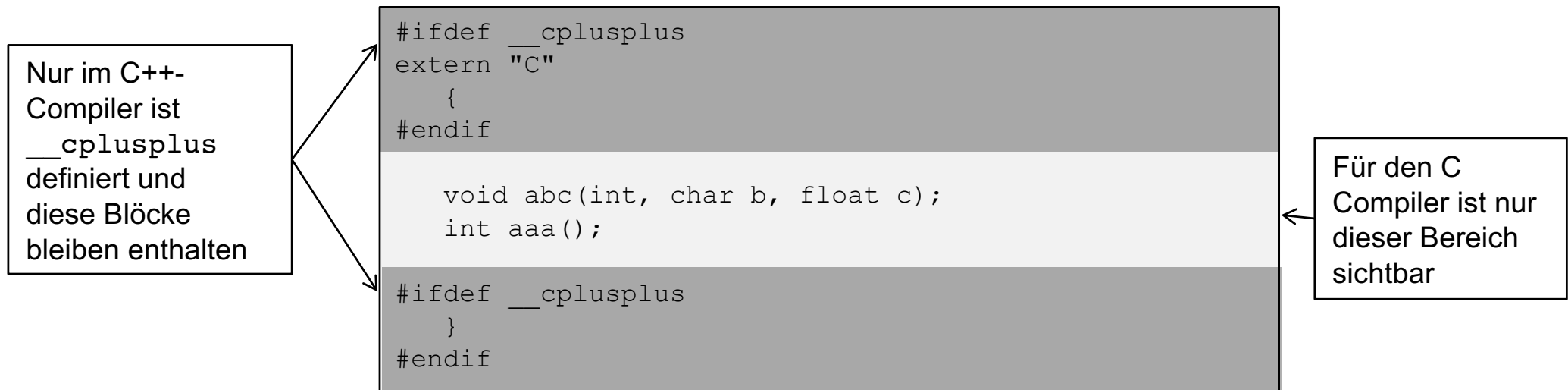
extern "C" Deklaration
einer einzelnen Funktion

```
extern "C"  
{  
    void abc(int, char b, float c);  
    int aaa();  
}
```

extern "C" Deklaration eines
Blocks

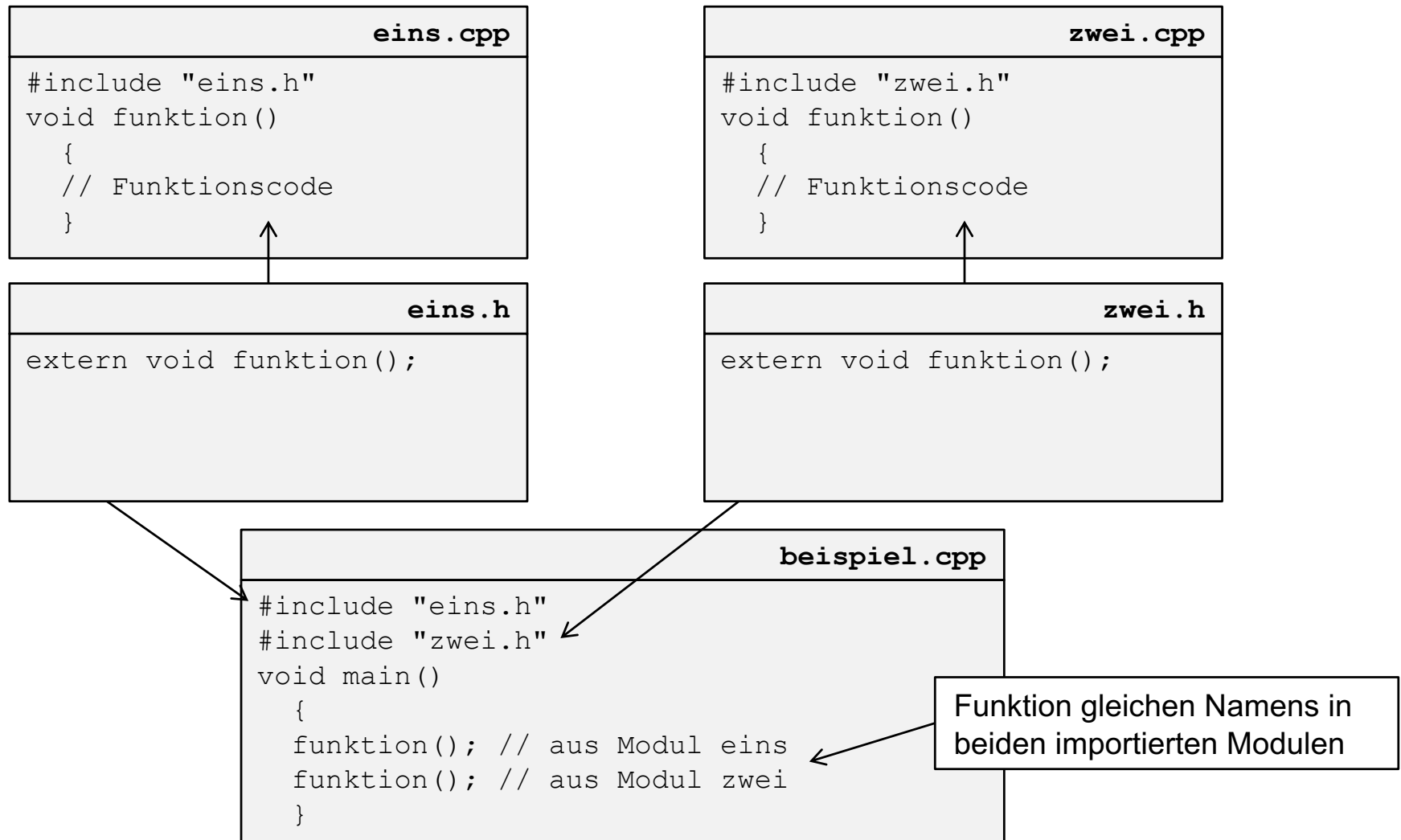
Einbinden von C-Funktionen in C++

Wenn wir in einem Header-File Funktionen als extern "C" deklarieren, stehen wir vor der Schwierigkeit, dass diese Deklaration in C nicht bekannt ist und auch gar nicht bekannt sein soll. Binden wir einen solchen Header in ein C-Programm ein, bedeutet dies dort einen Fehler. Wir müssen also verhindern, dass die entsprechende Zeile für den C-Compiler sichtbar wird. Dazu bedienen wir uns einen gängigen Tricks unter Einsatz des Präprozessors:



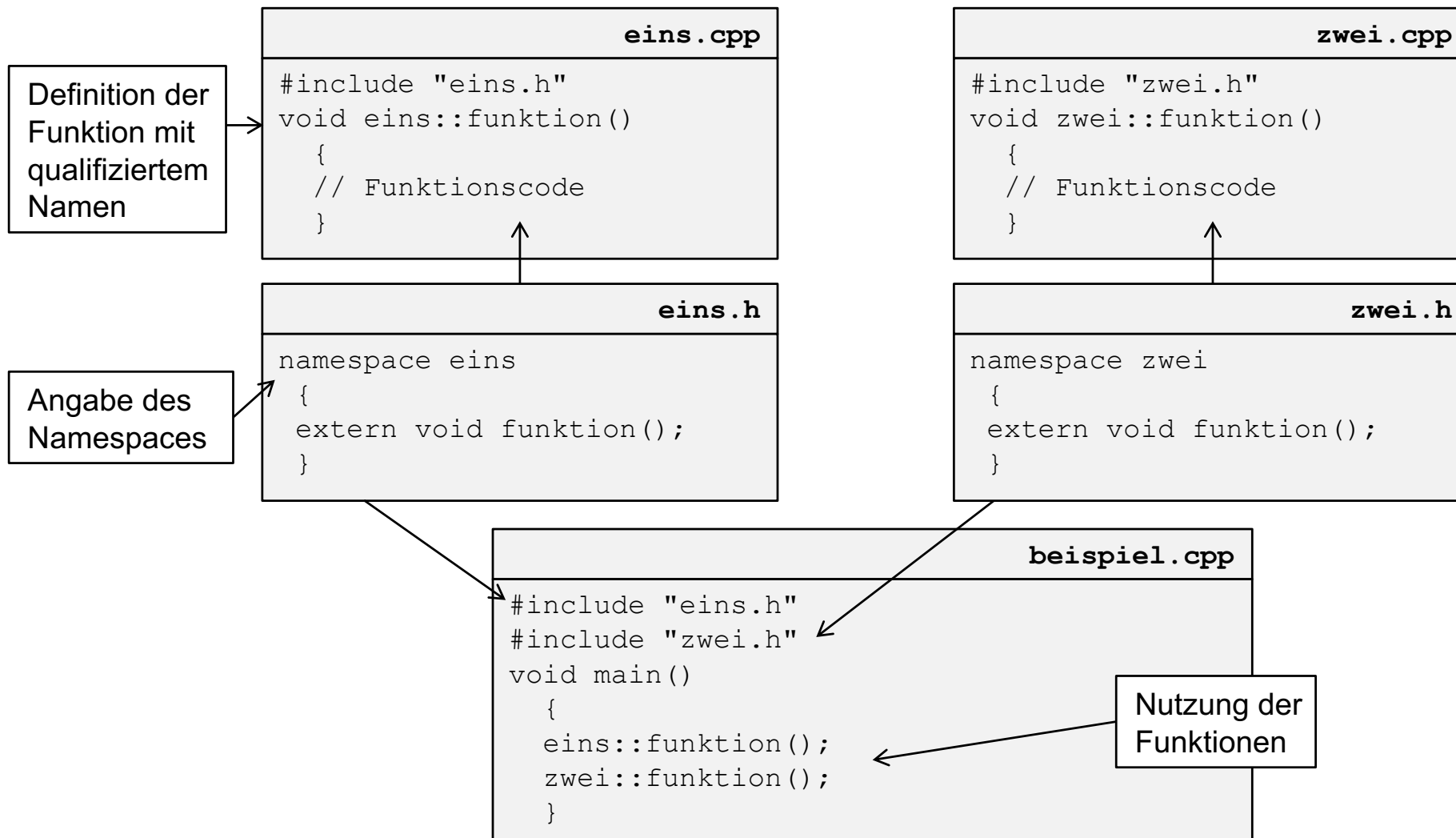
Auflösung von Namenskonflikten

Gerade bei großen Projekten kann es leicht vorkommen, dass Funktionen in unterschiedlichen Teilen des Projektes die gleichen Namen tragen. In diesem Fall kommt es zu Namenskonflikten. Ein Beispiel für einen solchen Fall sieht folgendermaßen aus:



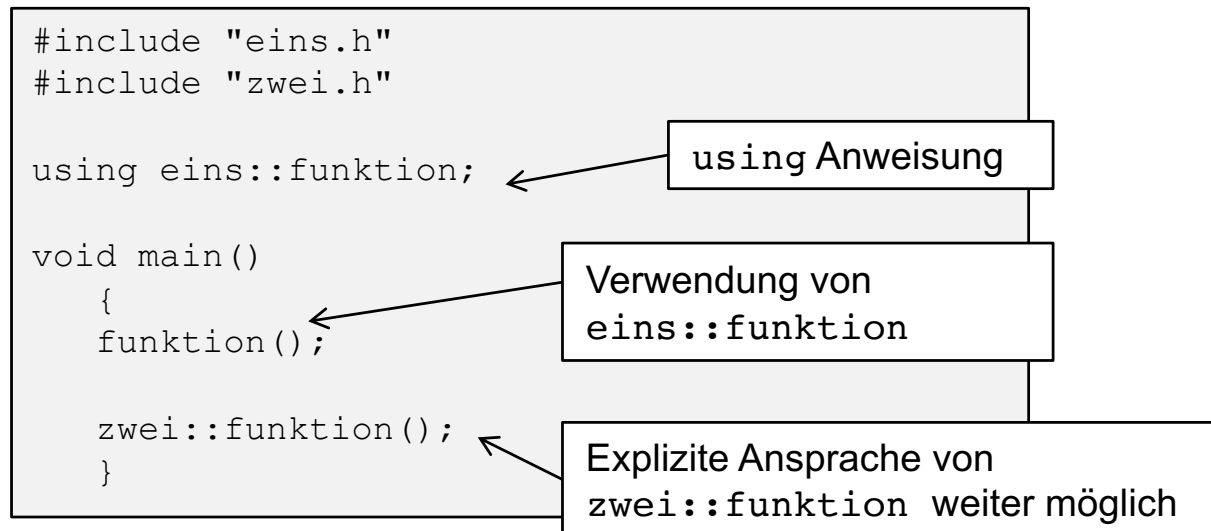
Namensräume in C++

In C++ können Funktionen in eigenen Namensräumen deklariert werden. Die Definition und Verwendung der Funktionen erfolgt dann unter Abgabe des vollständigen Namens (voll qualifiziert).



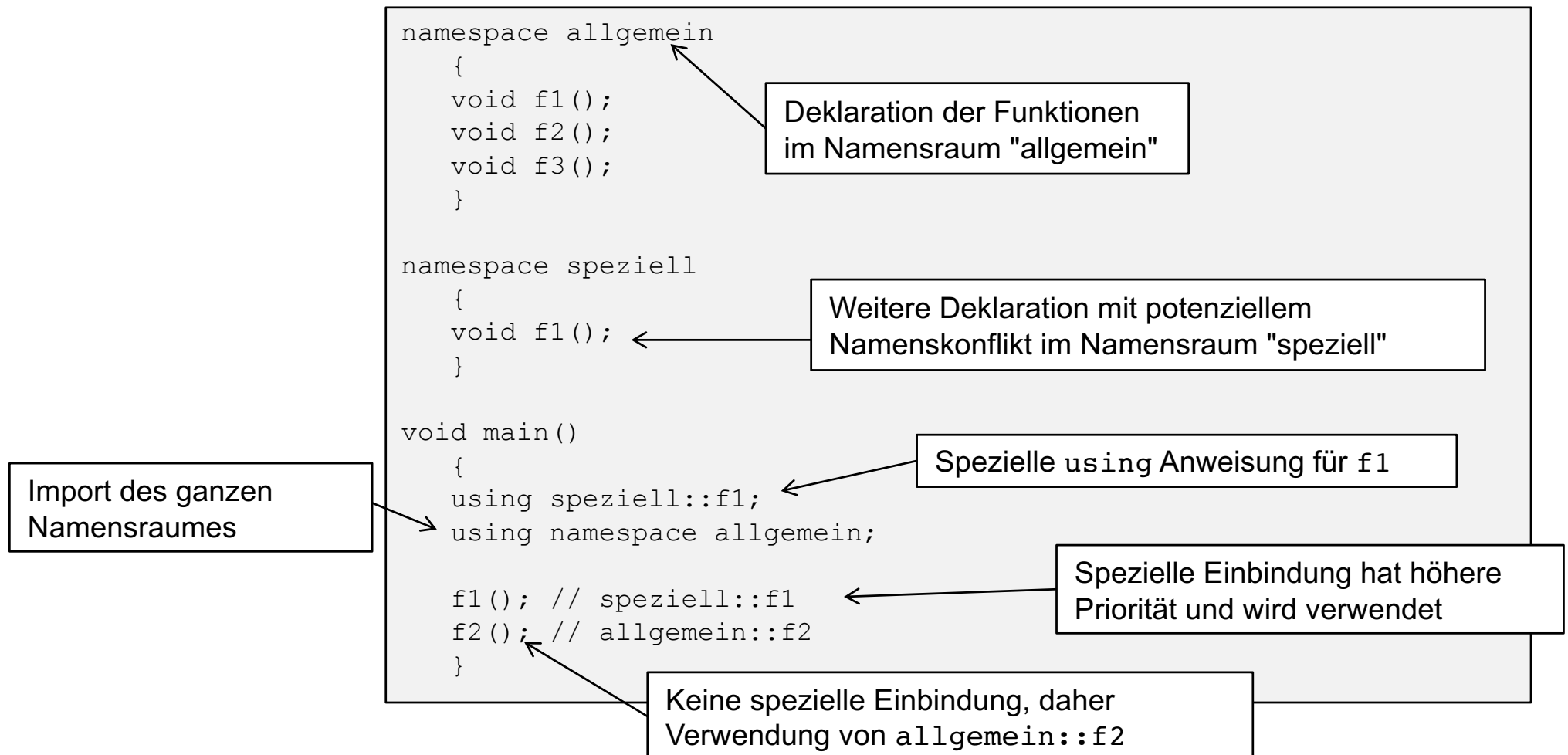
Ansprechen von Namensräumen

Die Angabe eines voll qualifizierten Namens wird bei langen Namen schnell unübersichtlich. Alternativ kann mit der `using` Anweisung mit ihrem voll qualifiziertem Namen eingebunden werden und danach mit dem kurzen Namen genutzt werden. Es sind mehrere `using` Anweisungen möglich, damit kann ein aufgelöster Namenskonflikt aber auch wieder eingeführt werden.



Importieren eines kompletten Namensraumes

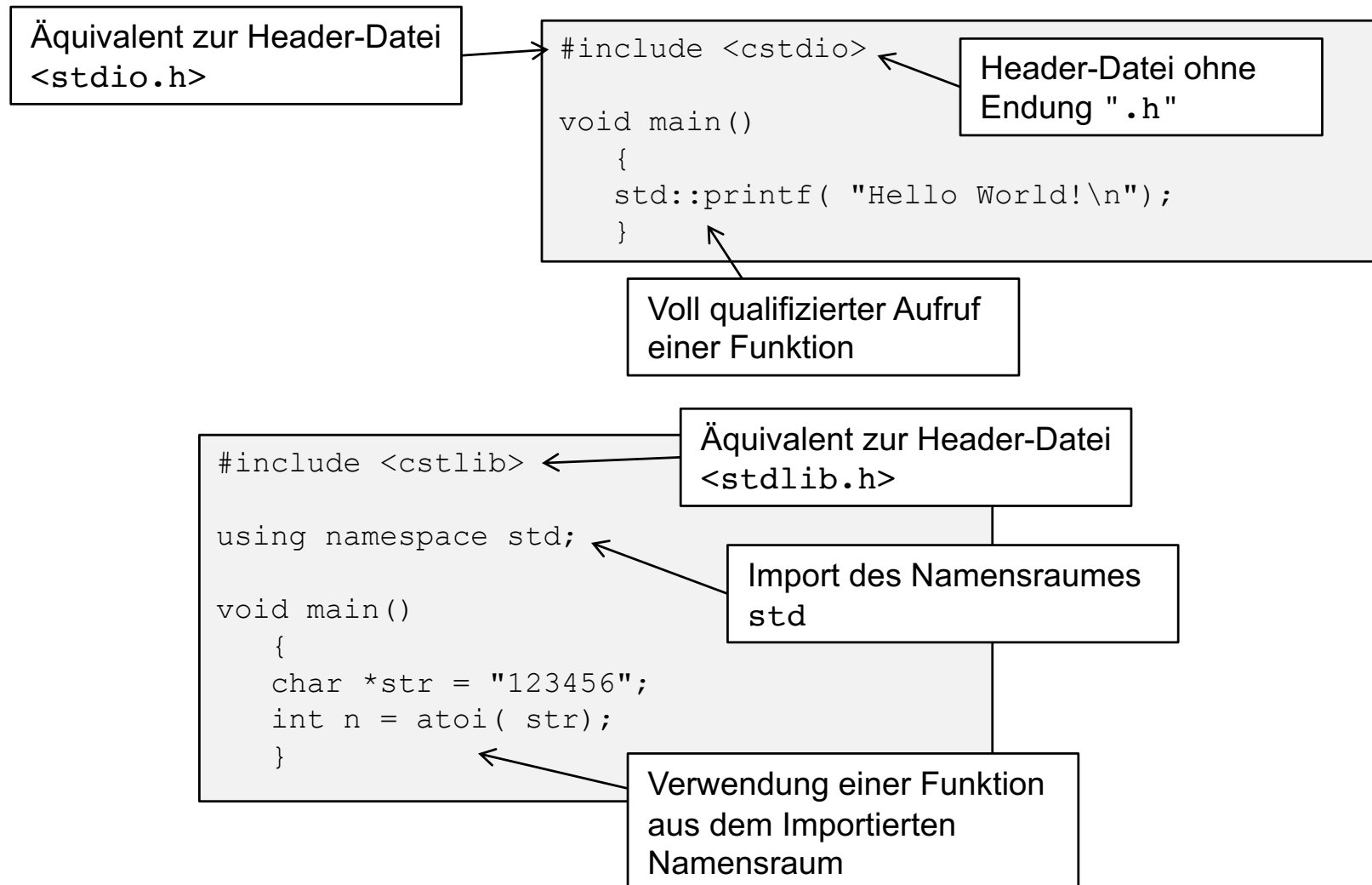
Über die `using` Anweisung ist es nicht nur möglich, einzelne Funktionen zu importieren, sondern einen vollständigen Namensraum als ganzes.



Der Standard Namensraum std

Die C-Laufzeitbibliotheken wurden für C++-Compiler mit angepassten Header-Dateien versehen. Deren Funktionsnamen sind dem Namensraum `std` zugeordnet.

Um Konflikte mit bestehenden Code zu vermeiden sind mehrere Nutzungen möglich:



Neue Schlüsselwörter für bekannte Operatoren

Für einige Operatoren, die wir bereits kennen, wurden neue Schlüsselwörter hinzugefügt, die ohne die entsprechenden Sonderzeichen auskommen, beispielsweise für Tastaturen, auf denen die entsprechenden Zeichen nur schwer zu erreichen sind oder für gänzlich andere Zeichensätze, die nicht alle Sonderzeichen enthalten.

Operator	Alternative
!	not
&&	and
&	bitand
	or
	bitor
^	xor
~	compl
!=	not_eq
=	or_eq
^=	xor_eq
&=	and_eq