



Begriffe

Der Begriff Objektorientierung (abgekürzt: OO) ist seit einigen Jahren sehr populär.

Man muss dabei unterscheiden zwischen

- **Objektorientierter Programmierung** (abgekürzt: **OOP**), das sind Programmier Techniken und -mittel, mit denen man objektorientierte Programme schreiben kann.
- **Objektorientierter Analyse** (abgekürzt: **OOA**), das ist eine auf Objektorientierung zugeschnittene Art der Problemanalyse.
- **Objektorientiertem Design** (abgekürzt: **OOD**), hierunter versteht man Entwurfs- und Modellierungsmethoden, die bei der Erstellung von objektorientierten Lösungen zu einer Problemstellung hilfreich sein können.



Objektorientierte Programmierung

Klassifikation von Programmiermodellen

Eine Programmiersprache ist eine formale Sprache zur Formulierung von Arbeitsanweisungen an ein Rechnersystem. Durch die Programmiersprache werden der Wortschatz (Anweisungen, Funktionen,...) und die Grammatik (Syntax dieser Anweisungen, Funktionen,...) eindeutig definiert, in der ein korrekter Programmtext zu schreiben ist. Höhere Sprachen werden mit Übersetzern in niedere übertragen.

Die höheren Sprachen werden in vier Kategorien unterteilt:



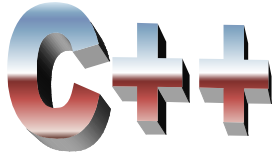
Objektorientierte Programmierung

Klassifikation von Programmiermodellen

- ***imperativ (prozedural):***

Bei imperativen (prozeduralen) Programmiersprachen besteht ein Programm (Programmrumpf) aus einer Abfolge von Operationen, die jeweils Daten bearbeiten. Wesentlich ist bei ihnen das Variablenkonzept, nach dem Eingabewerte in Variablen (Speicherzellen) abgelegt und dann weiterverarbeitet werden. Zu ihnen gehören Programmiersprachen wie:

ADA, BASIC, C, COBOL, FORTRAN, MODULA-2, PL/1 und PASCAL



Objektorientierte Programmierung

Klassifikation von Programmiermodellen

- ***deklarativ:***

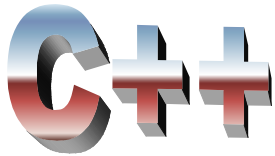
Deklarative Programmiersprachen gehen davon aus, dass vorrangig nicht die Problemlösung, sondern das Problem beschrieben wird, das zu lösen ist. Für diese Problembeschreibung gibt es verschiedene Ansätze.

- ***Funktionale Programmiersprachen:***

Das Ziel besteht darin, die Programmierung weitest möglich an die Formulierung mathematischer Funktionen anzunähern. Beispiele für funktionale Sprachen sind LISP und LOGO.

- ***Logische (prädikative) Programmiersprachen:***

Diese basieren auf der Prädikatenlogik. Mit einem Programm wird beispielsweise versucht, die Richtigkeit einer Eingabe anhand vorhandener Fakten und Regeln zu überprüfen. Die bekannteste logische Programmiersprache ist PROLOG.



Objektorientierte Programmierung

Klassifikation von Programmiermodellen

- **objektorientiert:**

Hier werden alle zum Problemlösen erforderlichen Informationen (Daten und Operationen) als Objekte aufgefasst (Objektorientierung!). Objekte sind gleichberechtigte, aktiv handelnde Einheiten, die miteinander kommunizieren, in dem sie Botschaften senden und empfangen. Im Bereich der objektorientierten Programmiersprachen sind zwei Richtungen vorhanden:

- **Rein objektorientierte Programmiersprachen:**

Rein objektorientierten Sprachen wurden von vornherein mit diesem Konzept entwickelt (SMALLTALK, EIFEL und JAVA)

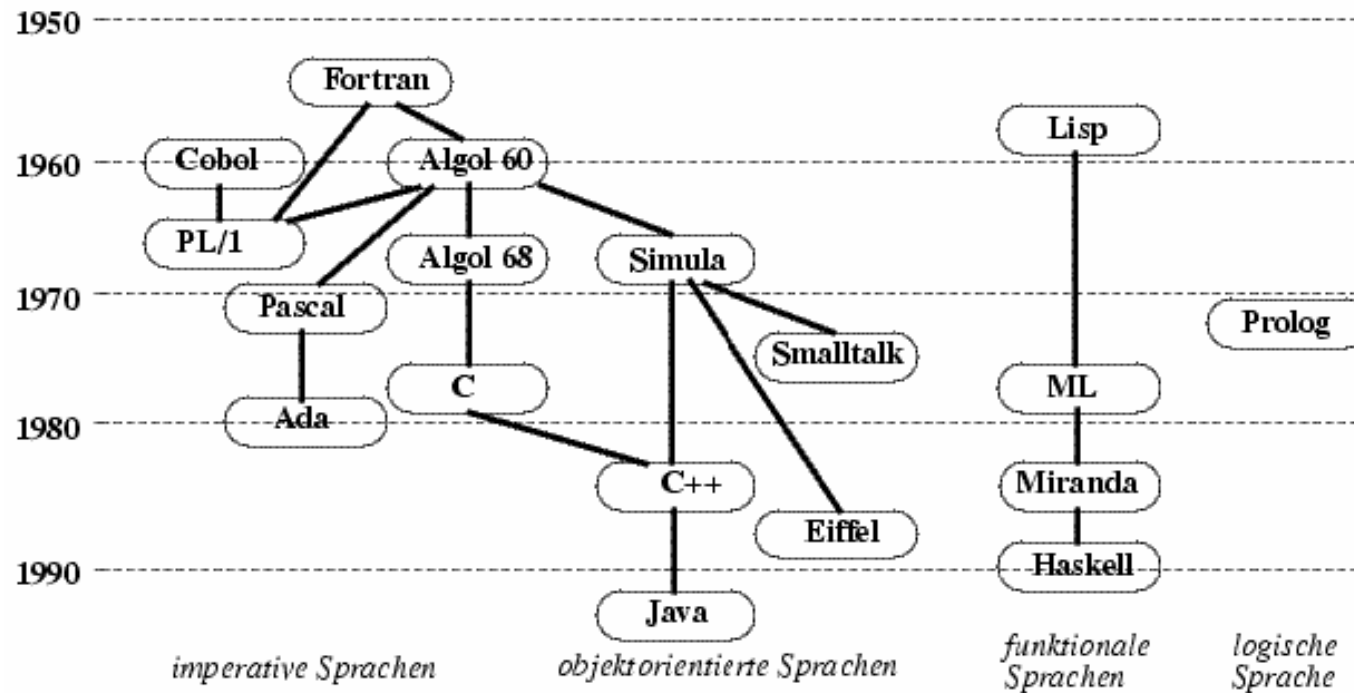
- **Hybridsprachen:**

Hybridsprachen sind Sprachen, die um die Möglichkeiten der objektorientierten Programmiersprachen erweitert wurden, wie z. B. C++, Object PASCAL und OBERON (Erweiterung von MODULA-2).

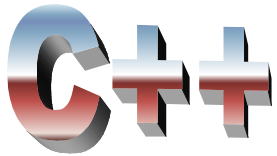


Objektorientierte Programmierung

Entstehungszeit und Verwandtschaft wichtiger Programmiersprachen



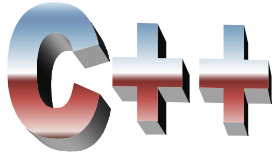
nach D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5



Objektorientierte Programmierung

Historie von C++:

1972	Entwicklung von C durch Dennis Ritchie und Ken Thompson zur Implementierung von Unix
1980	erste Version, als "C mit Klassen" entstanden bei AT&T Bell Laboratories entwickelt von Bjarne Stroustrup
1983	C++
1984	Einberufung eines ANSI-Komitees zur Standardisierung der Sprache
1987	Definition des ersten Standards durch das Buch von Stroustrup
Dezember 1996	ANSI-Draft (Aktuelle Version C++3.0)
November 1997	Verabschiedung des endgültigen ANSI-Standards



Objektorientierte Programmierung

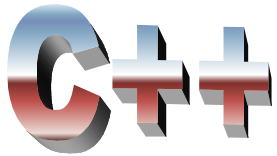
Vorteile gegenüber C

- ***Datenabstraktion:***

Klassen in C++ erlauben die Definition von Datenstrukturen, deren interne Details verborgen bleiben. Dies schützt die Inhalte der Strukturen nicht nur vor unsachgemäßem Umgang mit ihnen, sondern ermöglicht es auch, die interne Darstellung zu verändern, ohne dass andere Teile des Programms davon beeinflusst werden.

- ***Erweiterbarkeit:***

C++ erlaubt die Erweiterung der Sprache selbst durch die Definition von neuen Datentypen (in Form von Klassen), die von den fest eingebauten kaum unterscheidbar sind, weil sie so gut integriert sind. Gutes Beispiel dafür sind neue numerische Datentypen wie z.B. `complex`, auf die man die normalen numerischen Operatoren anwenden kann, oder auch `string`, denen man einfach eine Zeichenkette beliebiger Länge zuweisen kann, ohne dass sie überlaufen.



Objektorientierte Programmierung

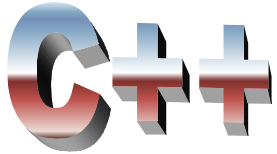
Vorteile gegenüber C

- **Programmieren im großen Stil:**

Die Unterstützung beim Schreiben großer Programme ist deutlich besser als bei C. Insbesondere lässt sich ein Programm als eine Sammlung von Klassen leichter beherrschen und warten.

- **Wiederverwendbarkeit von Code:**

Es ist in C++ leichter als in C, wieder verwendbare Komponenten zu schreiben. Klassen kann man vom Hauptprogramm gut trennen und in Klassenbibliotheken ablegen.



Objektorientierte Programmierung

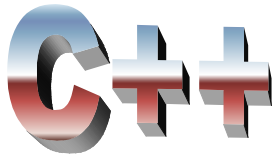
Vorteile gegenüber C

- **Objektorientierte Programmierung:**

C++ ist eine echte objektorientierte Sprache, die alles hierzu notwendige kennt: Klassen, Methoden, Polymorphismus, Vererbung.

- **Sicherheit:**

C++ ist viel strenger als C. Beispielsweise müssen Funktionen vor ihrer Verwendung deklariert werden. Diese erhöhte Sicherheit erlaubt dem Compiler, viele Fehler zu erkennen, die in C unentdeckt blieben.



Objektorientierte Programmierung

Nachteile von C++

- **Komplexität:**

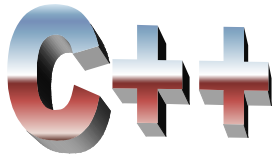
Die Sprache C mit ihrer Standardbibliothek ist schon nicht gerade übersichtlich. C++ enthält all dieses und noch sehr viel mehr. Außerdem können die ganzen Eigenschaften von C++ auch noch auf vielfältige Weise miteinander kombiniert werden.

- **Geringere Übersetzungsgeschwindigkeit:**

Da der Compiler viel aufwendiger ist, dauern die Verarbeitung des Quellcodes und die Erzeugung des Objectcodes auch entsprechend länger. Komplexere Compiler sind übrigens auch fehlerträglicher.

- **Fallen:**

Jede Sprachen hat ihre Tücken. Einige der Tücken von C wurden vermieden. Andere blieben erhalten, beispielsweise die ganze Zeigerarithmetik.



Objektorientierte Programmierung

Die wichtigsten Veränderungen/Erweiterungen gegenüber C

- **Kommentar**

//... zusätzlich zu /* ... */

- **Deklarationsreihenfolge:**

Variablen können an beliebiger Stelle im Code deklariert werden
– am besten dort, wo sie gleichzeitig mit einem Wert initialisiert werden können.

- **Arrays, Strukturen, Unions** usw. können in C++ mit beliebigen Werten initialisiert werden, auch mit Funktionswerten, Ausdrücken usw.

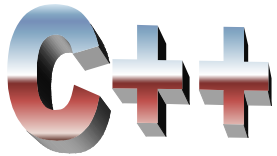
```
int n = 2;
```

```
int powers [] = {1, n, n*n, n*n*n, n*n*n*n};
```

- **echte Konstanten:**

```
const int n = 10;
```

```
int a[n]; // nur in C++
```



Objektorientierte Programmierung

Die wichtigsten Veränderungen/Erweiterungen gegenüber C

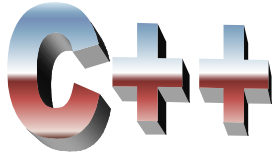
- **Ein-Ausgabe**

kann man in C++ wahlweise mit `<stdio.h>` oder mit `<iostream.h>` durchführen.

...aber nicht beides im selben Programm mischen!

```
cout << "Werte fuer i und j eingeben: ";  
cin >> i; cin >> j;  
//oder: cin>>i>>j;  
cout << "Summe ist " << i+j << endl;
```

Der Operator `<<` heißt Ausgabeoperator, weil er in einen Stream (Datenstrom, hier `cout`) etwas einfügt, der Operator `>>` heißt Eingabeoperator, weil er aus einem Stream (Datenstrom, hier `cin`) etwas herausholt/ extrahiert.



Objektorientierte Programmierung

Inkompatibilitäten zwischen C und C++

- **Die Größe von Zeichenkettenliteralen ist unterschiedlich.**
sizeof ('a') hat in C die Größe sizeof (int), in C++ aber die Größe sizeof (char).
- **In C kann eine Zeichenkette bei der Initialisierung**
randvoll geschrieben werden, ohne Platz für das ASCII-NULL-
Zeichen \0 zu lassen.
char Name [4] = "Hugo"; /* geht nur in C, nicht in C++ */
- **Typkonversionen**
in C nur mit cast-Ausdrücken, in C++ auch mit
Funktionsschreibweise.
x = (float) i; // in C und in C++
x = float(i); // nur in C++



Objektorientierte Programmierung

Warum mit objektorientierter Programmierung beginnen?

- Objektorientierter Programmentwurf ist nahe an der Problemanalyse und der Modellierung: Klassen und Objekte aus der Problemanalyse: objektorientierte Analyse, objektorientierter Entwurf
- Professionelle Software wird heute und wird in absehbarer Zukunft objektorientiert entwickelt.
- Objektorientierte Programmiersprachen haben mächtige Konzepte zur Abstraktion und Strukturierung: Klassen als abstrakte Datentypen, Vererbung zur Typspezialisierung und zur Schnittstellendefinition.



Objektorientierte Programmierung

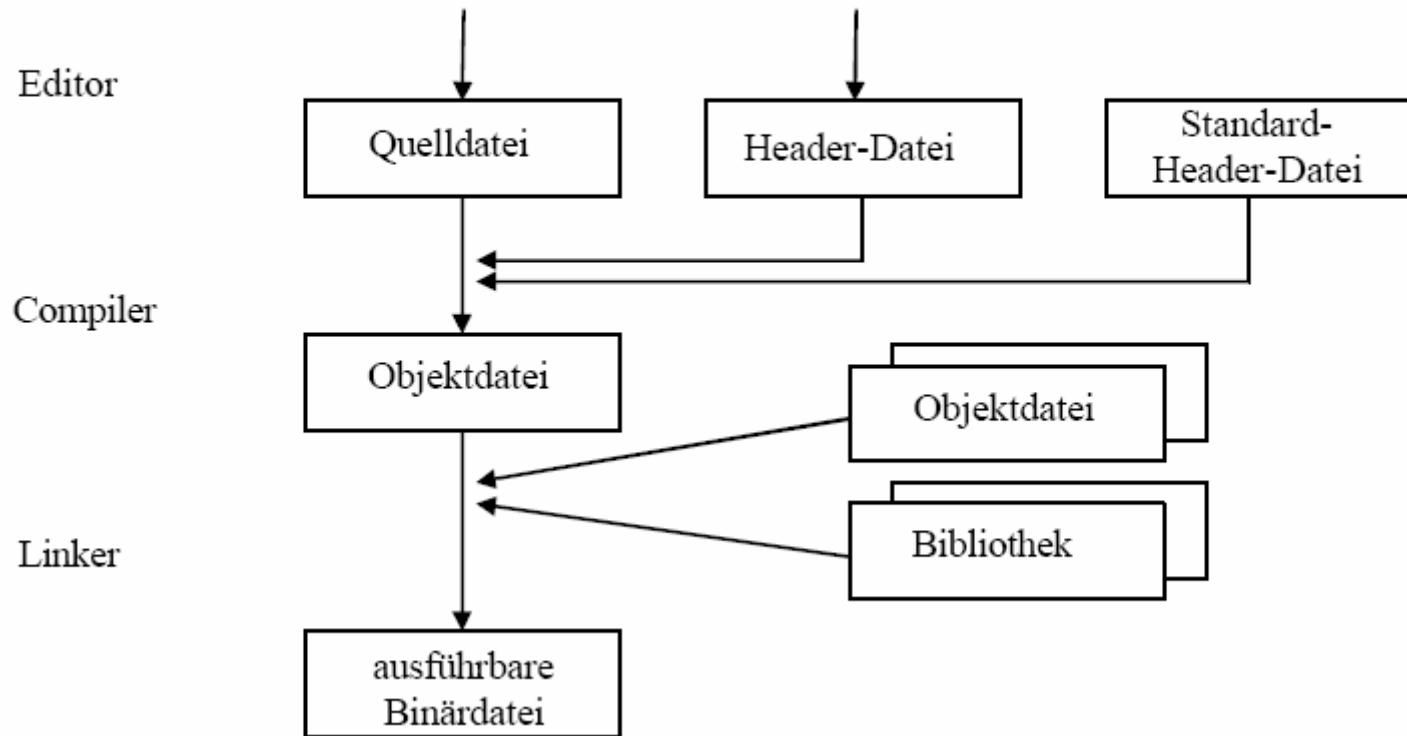
Warum mit objektorientierter Programmierung beginnen?

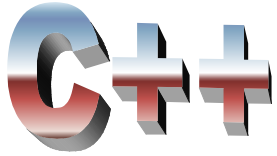
- Objektorientierte Bibliotheken mit wieder verwendbaren Komponenten
- Gut geplante Wiederverwendung steigert Produktivität bei der Software-Entwicklung, verbessert die Software-Qualität und vereinfacht die Wartung



Objektorientierte Programmierung

Entwicklung eines C++-Programmes



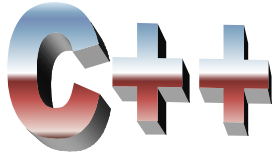


Objektorientierte Programmierung

Standard-Ein- und -Ausgabe in C++

Man kann in C++ ganz normal mit den Funktionen `scanf()` und `printf()` aus der C-Bibliothek `stdio.h` arbeiten, aber das ist nicht C++-typisch und nutzt die Vorteile von C++ nicht aus. Daher sollen hier die in C++ üblichen Ein- und Ausgabemechanismen erläutert werden – zunächst für die Konsole.

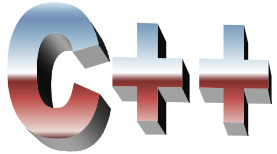
Hierzu ein kleines Beispielprogramm:



Objektorientierte Programmierung

Standard-Ein- und -Ausgabe in C++

```
#include <iostream>          // Includen der passenden Headerdatei
int min (int, int);         // Prototyp einer Minimum-Funktion
int main ()
{
    int x, y, z;
    cout << "x: "; cin >> x;
    cout << "y: "; cin >> y;
    cout << "die kleinere Zahl war: " << min (x, y) << endl;
    return 0;
}
int min (int a, int b)
{
    if (a < b) return a;
    return b;
}
```



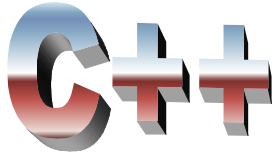
Objektorientierte Programmierung

Standard-Ein- und -Ausgabe in C++

Alle in C++ enthaltenen Datentypen können mit **cin >>** eingelesen und mit **cout <<** ausgegeben werden. Man muss sich dabei überhaupt nicht um Formatierung kümmern. Ausgaben kann man formatieren, aber dazu später mehr. Wir beschränken uns auf die Standardformatierung.

Mehrere Werte hintereinander einlesen oder auslesen kann man durch die mehrfache Verwendung des >> bzw. << Operators:

```
cin >> a >> b >> c;  
cout<<"Wert von a["<<i<<" ] ist "<<a [i]<< endl;
```



Objektorientierte Programmierung

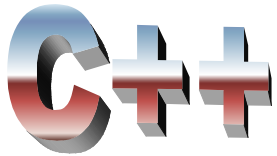
Standard-Ein- und -Ausgabe in C++

Das hier gezeigte **endl** schreibt ein Zeilenende und ist dem umständlichen "\n" vorzuziehen, weil es gleichzeitig auch alle Daten im Puffer ausgibt – entsprechend `fflush()` oder `flushall()` aus der CStandardbibliothek.

Um lediglich die Daten im Puffer auszugeben (ohne Zeilenvorschub), schreibt man

`cout.flush()` oder `cout << flush`

Zum Einlesen von Zeichenketten, die auch Leerzeichen enthalten können, d. h. ganzer Zeilen, dient die Methode `cout.getline()`



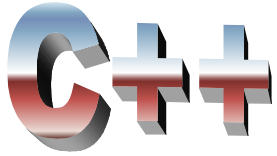
Objektorientierte Programmierung

Klassen und Kapselung

Die Klasse (**class**) ist der zentrale Begriff der objektorientierten Programmierung.

Die Klasse ist eine Erweiterung der aus der C-Programmierung bekannten Struktur (**struct**). Im folgenden werden zunächst nur zwei markante neue Eigenschaften betrachtet:

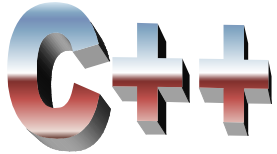
- Die Daten-Elemente einer Klasse sind per Voreinstellung **private**, auf sie kann nicht (wie auf die public voreingestellten Daten einer Struktur) direkt zugegriffen werden. Obwohl es für die Elemente einer Klasse Voreinstellungen hinsichtlich ihrer Eigenschaften gibt, werden trotzdem stets die Schlüsselwörter **private**, **protected** und **public** verwendet. Diese Schlüsselwörter können beliebig oft in der Deklaration einer Klasse verwendet werden, sie gelten jeweils bis zum Auftreten des nächsten Schlüsselwortes.



Objektorientierte Programmierung

Klassen und Kapselung

- Neben den Daten enthält eine Klasse auch Funktionen, mit denen die Daten der Klasse manipuliert werden können. Diese zur Klasse gehörenden Funktionen werden als **Methoden** bezeichnet (es sind auch die Begriffe "Elementfunktionen", "Schnittstellenfunktionen" bzw. "Member-Functions" gebräuchlich). Die Methoden sind per Voreinstellung public, können also aus allen Programmteilen aufgerufen werden, für die die Klassendeklaration "sichtbar" ist.

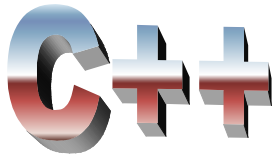


Objektorientierte Programmierung

Klassen und Kapselung

Hier wird zunächst eine besonders wichtige Eigenschaft der Klassen betrachtet: Man kann die in einer Klasse definierten Daten **kapseln**, sie gewissermaßen vor dem direkten Zugriff des Programmierers schützen, der sie **nur über die zur Klasse gehörenden Methoden manipulieren** kann.

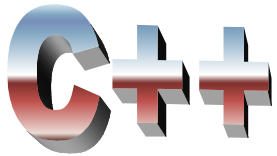
Diese Art der Programmierung ist zunächst aufwendiger, aber auch deutlich sicherer und vor allem wartungsfreundlicher.



Objektorientierte Programmierung

Definition einer Klasse

```
class CAnyname
{
    private:
        int a;
        float x;
        char s[200];
    public:
        void fkt1 (...)
        { ....}
        void fkt2 (...)
        { ....}
        void fkt3 (...)
        { ....}
};
```

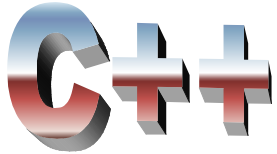


Objektorientierte Programmierung

Definition einer Klasse

In der Regel enthält eine Klassendefinition lediglich die Deklaration der Methoden (Funktionsprototypen), die Definition der Methoden findet außerhalb der Klasse statt.

Den Namen der Methoden muss dabei der Klassenname und der sog. **Scope-Operator** `::` vorangestellt werden, so dass der Compiler erkennt, zu welcher Klasse die Methode gehört:



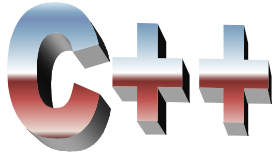
Objektorientierte Programmierung

Definition einer Klasse mit Prototypen

```
class CAnyname
{
    private:
        int a;
        float x;
        char s[200];
    public:
        void fkt1 (...);
        void fkt2 (...);
        void fkt3 (...);
};
void CAnyname::fkt1 (...){ }
```

```
void CAnyname::fkt2 (...){ }
```

```
void CAnyname::fkt3 (...){ }
```



Objektorientierte Programmierung

Definition einer Klasse

Die Klassendefinition wird in der Regel in einer eigenen Header-Datei gespeichert.

Vor dem Hauptprogramm wird diese dann mit include eingebunden:

```
# include "..."  
# include "Anyname.h"  
  
void main()  
{    ...    }
```

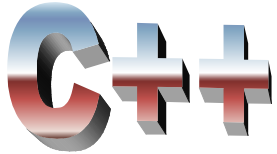


Objektorientierte Programmierung

Deklaration eines Objektes

Ein Objekt verhält sich zur Klasse, wie eine Variable zu ihrem Datentyp. Ebenso erfolgt die Deklaration eines Objektes:

```
# include "..."  
# include "Anyname.h"  
  
void main()  
{   CAnyname any1;  
    ...  
}
```

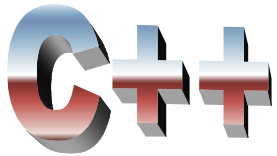


Objektorientierte Programmierung

Zugriff auf die Members eines Objektes

Auf sämtliche public definierte Members in der Klasse kann man (wie bei Strukturen) über den Objektnamen und den Punkt als Zugriffsoperator zugreifen:

```
# include "..."  
# include "Anyname.h"  
void main()  
{   CAnyname any1;  
    any1.fkt1();  
    any1.fkt2();  
    any1.a=0; // nicht erlaubt - Fehlermeldung!  
}
```



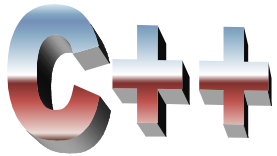
Objektorientierte Programmierung

Konstruktoren und Destruktoren

Beim Erzeugen einer Instanz einer Klasse wird immer eine spezielle Funktion aufgerufen, der sogenannte **Konstruktor**. Dieser wird entweder vom Compiler automatisch bereitgestellt ("Standard-Konstruktor"), kann (und sollte!) jedoch vom Programmierer geschrieben werden.

Konstruktoren können mehrfach überladen werden, d.h. mit unterschiedlichen Parameterlisten

Als Pendant dazu existiert immer ein **Destruktor**, der beim Erlöschen der Gültigkeit einer Instanz (z. B. am Ende des Blockes, in dem sie erzeugt wurde) aufgerufen wird.

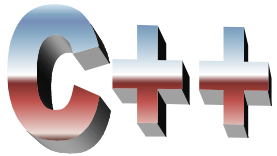


Objektorientierte Programmierung

Konstruktoren und Destruktoren

Regeln für das Schreiben von Konstruktor und Destruktor:

- Der Name des Konstruktors muss immer mit dem Namen der Klasse identisch sein (der Compiler unterscheidet auf diese Weise den Konstruktor von den übrigen zur Klasse gehörenden Methoden).
- Der Name des Destruktors muss aus dem Namen der Klasse mit einer vorangestellten Tilde ~ gebildet werden.
- Der Konstruktor kann Argumente übernehmen (dies ist der Regelfall, da er vorwiegend zur Initialisierung von Daten der Klasse benutzt wird). Ein Destruktor kann grundsätzlich keine Argumente übernehmen.
- Weder Konstruktor noch Destruktor liefern einen Return-Wert zurück, auch nicht void!
- Default-Konstruktor: Falls überhaupt kein Konstruktor definiert wurde



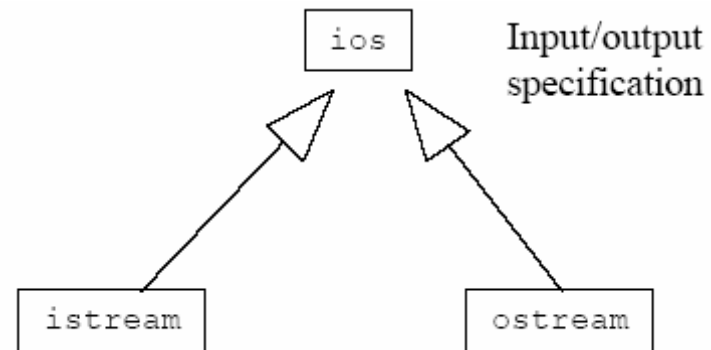
Objektorientierte Programmierung

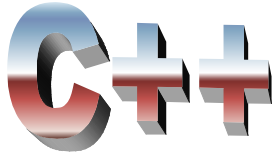
Ein-/Ausgabe - Die iostream-Bibliothek

Die iostream-Bibliothek ist komplett objektorientiert aufgebaut und bietet Datentypen in Form von Klassen für die folgenden Aufgaben:

- Ein-/Ausgabe auf die Standardeingabe bzw -ausgabe
- Dateiverarbeitung
- Einlesen aus und Ausgeben in Zeichenketten
- Fehlerbehandlung

Die wichtigsten Klassen der iostream-Bibliothek sind die folgenden:





Objektorientierte Programmierung

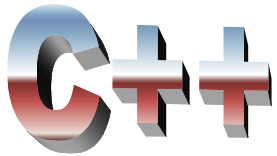
Vordefinierte iostream-Objekte

Die folgenden iostream-Objekte sind bereits definiert, wenn ein Programm gestartet wird:

- cin Standardeingabe (istream)
- cout Standardausgabe (ostream)
- cerr Standardfehlerausgabe (ostream, ungepuffert)
- clog Standardfehlerausgabe (ostream, gepuffert), (wird zur Fehlerausgabe in Dateien benutzt)

Die Ein-/Ausgabe kann über die Operatoren >> bzw. << erfolgen oder über spezielle Elementfunktionen.

Für alle Standarddatentypen sind die Operatoren << und >> vordefiniert.



Objektorientierte Programmierung

Ein-Ausgabe der elementaren Datentypen

Ausgabe:

```
cout << "hallo"; // gibt die Zeichenkette "hallo" auf die
                // Standardausgabe aus
cout << "Wert:" << j; //gebe "Wert:" aus und dann j
cout << a + b; //gebe Summe a+b aus
```

Eingabe:

```
cin >> i;
cin >> i >> j; // lese zuerst i und dann j ein
```

Einlesen von Zeichen:

Der Operator >> liest nur das nächste nichtleere Zeichen aus dem Eingabestrom.

```
char c;
cin >> c; // liest das nächste nichtleere Zeichen
```



Objektorientierte Programmierung

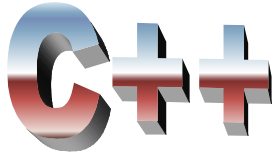
Ein-Ausgabe der elementaren Datentypen

Sollen auch Leerzeichen gelesen werden, so kann dazu die spezielle Methode `get()` verwendet werden.

```
char c;  
cin.get(c); // lese nächstes Zeichen in c ein
```

Das Gegenstück zu `get()` bei `cout` heißt `put()` und gibt ein Zeichen auf die Standardausgabe aus.

```
char c;  
while (cin.get(c))  
cout.put(c); // ein Zeichen ausgeben
```



Objektorientierte Programmierung

Einfache Fehlerbehandlung

Beispiel: Programm erwartet int-Wert:

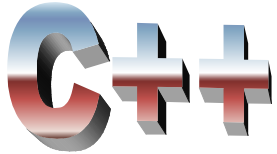
```
int i;  
cin >> i;
```

Aber der Benutzer gibt ein:

```
Hallo
```

Folge: Das Programm liest keinen sinnvollen ganzzahligen Wert

- iostream-Objekte protokollieren ihren Zustand mit. Dieser Zustand kann über spezielle Elementfunktionen abgefragt werden.
- Jedes iostream-Objekt verwaltet seinen Zustand getrennt und unabhängig von den anderen.
- Im obigen Beispiel merkt sich cin, daß eine Eingabe misslungen ist. Diese Information kann nachher abgefragt werden. Das Programm kann diese Information auswerten und ggf. passend reagieren.



Objektorientierte Programmierung

Einfache Fehlerbehandlung

Folgendes Programmfragment erkennt Fehleingaben:

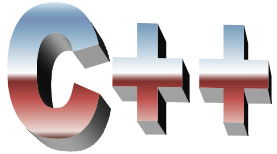
```
int i;
cin >> i;
if (cin.good()) // gleichbedeutend mit if (cin)
    cout << "Ok, " << i << " eingegeben" << endl;
else cout << "Eingabefehler" << endl;
```

Ein `istream`-Objekt merkt sich nur den Zustand der letzten I/O-Aktion. Der vorletzte und weiter zurückliegende Zustände werden nicht gespeichert.

Ein fehlerhafter Zustand bleibt so lange gespeichert, bis er ausdrücklich gelöscht wird. Dazu gibt es die folgende Elementfunktion:

```
cin.clear()
```

Wirkung: Das sogenannte `good`-Bit wird auf `true` gesetzt, eventuelle Fehlerbits auf `false`.



Objektorientierte Programmierung

Einfache Fehlerbehandlung

Problem: Falls kein korrekter Wert von cin gelesen werden konnte, bleiben die fehlerhaften Daten im Eingabestrom stehen und werden beim nächsten Eingabeversuch wieder gelesen.

Diese fehlerhaften Daten müssen also zunächst aus dem Eingabestrom entfernt werden.

Dazu gibt es folgende Elementfunktion:

```
cin.ignore(int n, char c)
```

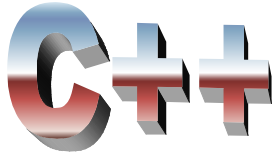
Wirkungsweise:

Es werden maximal n Zeichen überlesen. Das Zeichen c legt fest, bis zu welchem Zeichen im Eingabestrom die Eingabe zu verwerfen ist.

Üblicherweise nimmt man hier das Linefeedzeichen '`\n`'.

`ignore` liest also die Eingabe, bis

- entweder n Zeichen gelesen wurden
- oder ein Zeichen c auftaucht

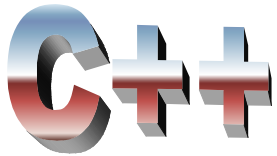


Objektorientierte Programmierung

Einfache Fehlerbehandlung

Im folgenden Beispiel werden maximal 80 Zeichen bzw. der Rest der Zeile ausgelassen.

```
int i;
cin >> i; // Erster Leseversuch
while(cin.fail()) // Test auf Misserfolg
{
    cin.clear(); // Misserfolg: Zustand zurücksetzen
    cout << "Fehleingabe - bitte wiederholen" << endl;
    cin.ignore(80, '\n'); // Fehleingabe überspringen
    cin >> i; // neuer Leseversuch
}
```



Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

Prinzipiell gibt es 2 Möglichkeiten, das Aus- bzw. Eingabeformat zu beeinflussen:

Elementfunktionen der Klassen `istream` bzw. `ostream` oder sogenannte Manipulatoren (`<iomanip>`), die in den Datenstrom eingefügt werden.

Feldbreite, Ausrichtung und Füllzeichen:

```
cout.width (10)           // Elementfunktion  
cout << setw(10)<< a;     // Manipulator
```

➔ setzt die Feldbreite zum Einlesen oder Ausgeben (wird automatisch zurückgesetzt)

```
cout.fill ('0');         // Elementfunktion  
cout << setfill('0')<< a; // Manipulator
```

➔ Füllzeichen setzen, Voreinstellung ' '



Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

Ausrichtung des Feldes

```
cout.setf(ios::left, ios::adjustfield)
```

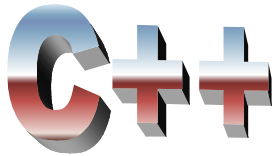
→ linksbündige Ausrichtung des auszugebenden Feldes

```
cout.setf(ios::right, ios::adjustfield)
```

→ rechtsbündige Ausrichtung des auszugebenden Feldes
(Voreinstellung)

```
cout.setf(ios::internal, ios::adjustfield)
```

→ Vorzeichen linksbündig, Wert rechtsbündig

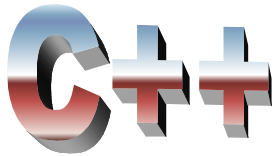


Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

Beispiel:

```
#include <iostream>
#include <iomanip>
void main()
{
int zahl = -17;
// Feldbreite 10 und Füllzeichen #
cout << setw(10) << setfill('#') << zahl << endl;    // #####-17
// linksbündig ausgeben
cout.setf (ios::left, ios::adjustfield);
cout << setw(10) << setfill('#')<< zahl << endl;    // -17#####
// Vorzeichen links und Wert rechts
cout.setf (ios::internal, ios::adjustfield);
cout << setw(10) << setfill('#')<< zahl << endl;    // -#####17
}
```



Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

Formatierung ganzer Zahlen

Einstellung von Ausgabeformaten:

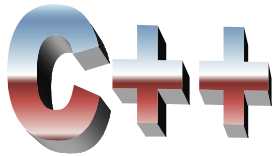
```
cout.setf (ios::showbase) //Ausgabe mit der Basis,  
                        //bei oktal fuehrende 0,  
                        //bei Hex fuehrendes 0x  
cout.setf (ios::showpos) //Ausgabe von positiven Vorzeichen  
cout.setf (ios::uppercase) //Ausgabe von Großbuchstaben,  
                        //z.B.bei Hex
```

Setzen der Zahldarstellung:

Elementfunktion

Manipulator

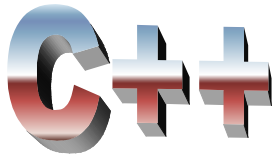
```
cout.setf (ios::hex) cout<<hex.. Hexadezimaldarstellung einschalten  
cout.setf (ios::oct) cout<<oct.. Oktaldarstellung einschalten  
cout.setf (ios::dec) cout<<dec.. Dezimaldarstellung einschalten
```



Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

```
void main() {
cout.setf (ios::showbase); // Ausgaben mit Anzeige der Basis
cout<<hex<<100<<" " <<oct<<100<<"\n\n";      //x64 0144
cout.setf (ios::uppercase);          // uppercase
cout<<hex<<100<<endl;                    //X64
cout.unsetf (ios::uppercase);        // lowercase
cout<<hex<<100<<"\n\n";                  //x64
cout.setf (ios::showpos); // Ausgaben mit pos. Vorzeichen
// Mit Manipulator:                // +100 0144 0x64
cout<<dec<<100<<" " <<oct<<100<<" " <<hex<<100<<endl;
// Mit Elementfunktion:
cout.setf(ios::dec); cout << 101 << " ";
cout.setf(ios::oct); cout << 101 << " ";
cout.setf(ios::hex); cout << 101 << endl; // +101 0145 0x65
}
```



Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

Formatierung von Gleitpunktzahlen

Einstellung des Ausgabeformats:

<code>cout.setf(ios::fixed)</code>	Dezimaldarstellung
<code>cout.setf(ios::scientific)</code>	Exponentialdarstellung
<code>cout.setf(ios::showpoint)</code>	Dezimalpunkt und abschließende Nullen immer darstellen

Einstellung der Genauigkeit:

Elementfunktion

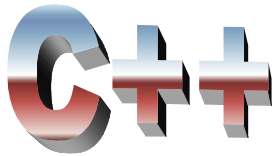
```
cout.precision(4);
```

Manipulator

```
cout << setprecision(4)...
```

setzt die Genauigkeit der Ausgabe, Voreinstellung ist 6

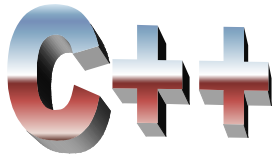
Bei der Ausgabe der Nachkommastellen wird gerundet ausgegeben, die Zahl selbst bleibt aber nach wie vor mit ihrer vollen Stellenzahl gespeichert



Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

```
#include <iostream>
#include <iomanip>
void main() {
cout<<"precision normal\n";
cout<<"2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<
    endl;          // 2: 3.2e+02    0.12
cout<<"6:\t"<<setprecision(6)<<321.0<<' \t '<<0.12345678<<
    endl;          // 6: 321    0.123457
cout <<"\nprecision showpoint\n";
cout.setf(ios::showpoint);
cout <<"2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<
    endl;          // 2: 3.2e+02    0.12
cout <<"6:\t"<<setprecision(6)<<321.0<<' \t '<<0.12345678<<
    endl;          // 6: 321.000    0.123457
```

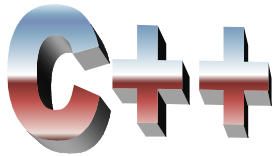


Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

```
cout<<"\nprecision fixed\n";  
cout.setf(ios::fixed,ios::floatfield);  
cout<<" 2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<  
endl;          // 2: 321.00  0.12  
cout<<" 6:\t"<<setprecision(6)<<321.0<<' \t '<<0.12345678<<  
endl;          // 6: 321.000000  0.123457
```

```
cout<<"\nprecision scientific\n";  
cout.setf(ios::scientific);  
cout<<" 2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<  
endl;          // 2: 3.21e+02  1.23e-0  
cout<<" 6:\t"<<setprecision(6)<<321.0<<' \t '<< 0.12345678 <<  
endl;          // 6: 3.210000e+02  1.234568e-01
```

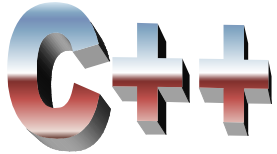


Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

```
cout<<"\nprecision fixed\n";
cout.setf(ios::fixed,ios::floatfield);
cout<<" 2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<
endl;          // 2: 321.00  0.12
cout<<" 6:\t"<<setprecision(6)<<321.0<<' \t '<<0.12345678<<
endl;          // 6: 321.000000  0.123457
```

```
cout<<"\nprecision scientific\n";
cout.setf(ios::scientific);
cout<<" 2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<
endl;          // 2: 3.21e+02  1.23e-0
cout<<" 6:\t"<<setprecision(6)<<321.0<<' \t '<< 0.12345678 <<
endl;          // 6: 3.210000e+02  1.234568e-01
```

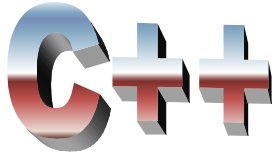


Objektorientierte Programmierung

Formatierungsmöglichkeiten bei Ein-/Ausgabe

```
cout<<"\nprecision fixed\n";
cout.setf(ios::fixed,ios::floatfield);
cout<<" 2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<
endl;          // 2: 321.00  0.12
cout<<" 6:\t"<<setprecision(6)<<321.0<<' \t '<<0.12345678<<
endl;          // 6: 321.000000  0.123457
```

```
cout<<"\nprecision scientific\n";
cout.setf(ios::scientific);
cout<<" 2:\t"<<setprecision(2)<<321.0<<' \t '<<0.12345678<<
endl;          // 2: 3.21e+02  1.23e-0
cout<<" 6:\t"<<setprecision(6)<<321.0<<' \t '<< 0.12345678 <<
endl;          // 6: 3.210000e+02  1.234568e-01
```



Objektorientierte Programmierung

friend-Funktionen und friend-Klassen

Es gibt bei C++ Situationen, wo es notwendig ist, dass eine globale Funktion auf private Daten eines Objektes direkt zugreifen muss. Ebenso kann es vorkommen, dass zwei Klassen so eng miteinander arbeiten, dass private Informationen unmittelbar zugänglich sein müssen. Um dieses Problem zu lösen, hat man in C++ das friend-Konzept geschaffen. Eine Funktion, die nicht Elementfunktion einer Klasse XY ist, kann dennoch auf private- oder protected- Merkmale von XY zugreifen, wenn sie als friend von X spezifiziert ist. Eine Klasse, die als friend-Klasse von X deklariert ist, hat ebenso das Recht auf private Merkmale zuzugreifen.

Alle friends müssen explizit in der Klassendefinition deklariert werden und sind damit als Bestandteil der Klassenschnittstelle zu betrachten. Eine Hauptanwendung von friends ist das Überladen von Operatoren.



Objektorientierte Programmierung

friend-Funktionen und friend-Klassen

Per Konvention schreibt man sie ganz an den Anfang einer Klasse. Die Definition erscheint oft außerhalb der Klasse. Das Schlüsselwort `friend` lässt man dort weg. Den Scope-Operator braucht man dort auch nicht, weil es sich ja nicht um ein Member handelt.

```
class X; // Vorwärtsdeklaration
class Y
{ public:
    void f(X* xp);
    void g(X* xp);
};
```

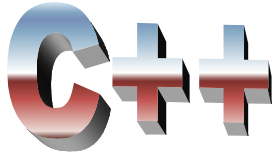


Objektorientierte Programmierung

friend-Funktionen und friend-Klassen

```
class Z
{ public:
    void f(X* xp);
    void g(X* xp);
};

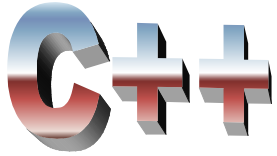
class X
{ friend class Y;
  friend void Z::f(X*);
  friend void f(X*);
  public:
    X(int a):i(a) {}
  private:
    int i;
};
```



Objektorientierte Programmierung

friend-Funktionen und friend-Klassen

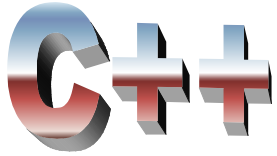
```
void Y::f(X* xp) { cout << "Y::f() " << xp->i << endl; }
void Y::g(X* xp) { cout << "Y::g() " << xp->i << endl; }
void Z::f(X* xp) { cout << "Z::f() " << xp->i << endl; }
void Z::g(X* xp) { cout << "Z::g() " << xp->i << endl; }
// Zugriff auf i nicht erlaubt
void f(X* xp) { cout << "f() " << xp->i << endl; }
```



Objektorientierte Programmierung

friend-Funktionen und friend-Klassen

```
void main()
{
    X* xp = new X(1);
    Y y1;
    Z z1;
    y1.f(xp);
    y1.g(xp);
    z1.f(xp);
    // z1.g(xp); kein friend
    f(xp);
}
```



Objektorientierte Programmierung

Operatorüberladung

Funktionsstyp **operator** <Operator-Symbol> (Parameterliste) {...}

C++ bietet die Möglichkeit, Operatoren für eigene Typen zu definieren, man könnte etwa für eine Klasse A:

```
class A { ... };
```

den Operator + definieren, so dass in einer Anwendung folgendes möglich wird:

...

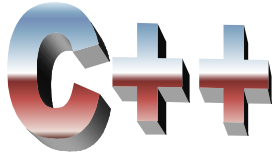
```
A a,b; // zwei A-Objekte
```

...

```
... a + b ... // "Addition" von a und b
```

...

d.h. man könnte mit selbstdefinierten Typen Ausdrücke (Verknüpfung von Operanden mit Operatoren) formulieren.

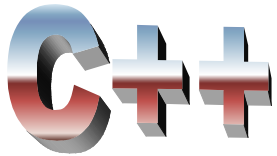


Objektorientierte Programmierung

Operatorüberladung

Definition einer Operatorüberlagerung:

```
class CTest
{ private:
    int data;
    public:
        CTest operator + (CTest);
        ...
};
CTest CTest ::operator + (CTest a)
{ CTest temp;
  temp.data = ...
  return temp;
}
```

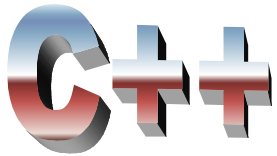


Objektorientierte Programmierung

Operatorüberladung

Folgende Operationszeichen kann man für eigene Datentypen (neu-)definieren:

<i>Operator</i>	<i>Bedeutung</i>
+ - * / %	Arithmetische Operatoren (- und + auch unär: -a, +a)
++ --	Inkrement und Dekrement (als Präfix (z.B.: ++a) und Postfix (z.B.: a++))
< > <= >= != ==	Vergleichsoperatoren
! &&	Logische Operatoren
^ & ###	Bitoperatoren
<< >>	Bitshiftoperatoren
= += -= *= /= %= ^= &= = <<= >>=	Zuweisungsoperatoren
[]	Indexoperator
()	Funktionsaufrufoperator
-> ->*	Zugriffsoperatoren
,	Kommaoperator
& *	Adress- und *-operator, z.B.: &a *ptr
new new[] delete delete[]	Speicher allokieren bzw. deallokieren

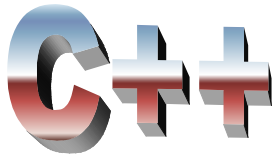


Objektorientierte Programmierung

Operatorüberladung

Die übrigen, in folgender Tabelle aufgeführten Operationszeichen kann man nicht überladen:

.	::	.*	?:	sizeof
---	----	----	----	--------



Objektorientierte Programmierung

Operatorüberladung

Regeln:

- Eine Operatorfunktion muss entweder Elementfunktion einer Klasse sein oder mindestens einen Parameter vom Typ Klasse oder Referenz auf Klasse haben.
- Operatorfunktionen können nicht für elementare Datentypen definiert werden.
- Die Anzahl der Operanden kann nicht verändert werden, d.h. binäre Operatoren bleiben binär, unäre Operatoren bleiben unär.
- Die Priorität eines Operators kann nicht verändert werden.
- Die Assoziativität, d. h. die Reihenfolge der Zusammenfassung z.B. von links nach rechts oder von rechts nach links, kann nicht verändert werden.
- Es können nur existierende Operatoren überladen werden, d.h. es können keine neuen Operatoren erfunden werden.
- Mit Ausnahme des Zuweisungsoperators (`operator=()`) werden alle Operatorfunktionen an abgeleitete Klassen weiter vererbt.
- Bei manchen Operatoren gibt es spezielle Vorschriften für Rückgabotyp und Argumente.