

# Schrödinger programmiert C++

Jetzt mit C++14 und Syntaxhighlighting

Bearbeitet von  
Dieter Bär

2., aktualisierte Auflage 2015. Buch. 696 S. Kartoniert  
ISBN 978 3 8362 3824 3  
Format (B x L): 20,2 x 23,6 cm  
Gewicht: 1632 g

[Weitere Fachgebiete > EDV, Informatik > Programmiersprachen: Methoden > Programmier- und Skriptsprachen](#)

schnell und portofrei erhältlich bei






  
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung [beck-shop.de](http://beck-shop.de) ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.



## Leseprobe

*Mit Schrödinger werden Sie fit in C++! Lernen Sie zusammen mit ihm alle Feinheiten von C++. Hier können Sie sich bereits die Grundlagen aneignen. Außerdem enthält diese Leseprobe das Inhaltsverzeichnis und das gesamte Stichwortverzeichnis des Buches.*

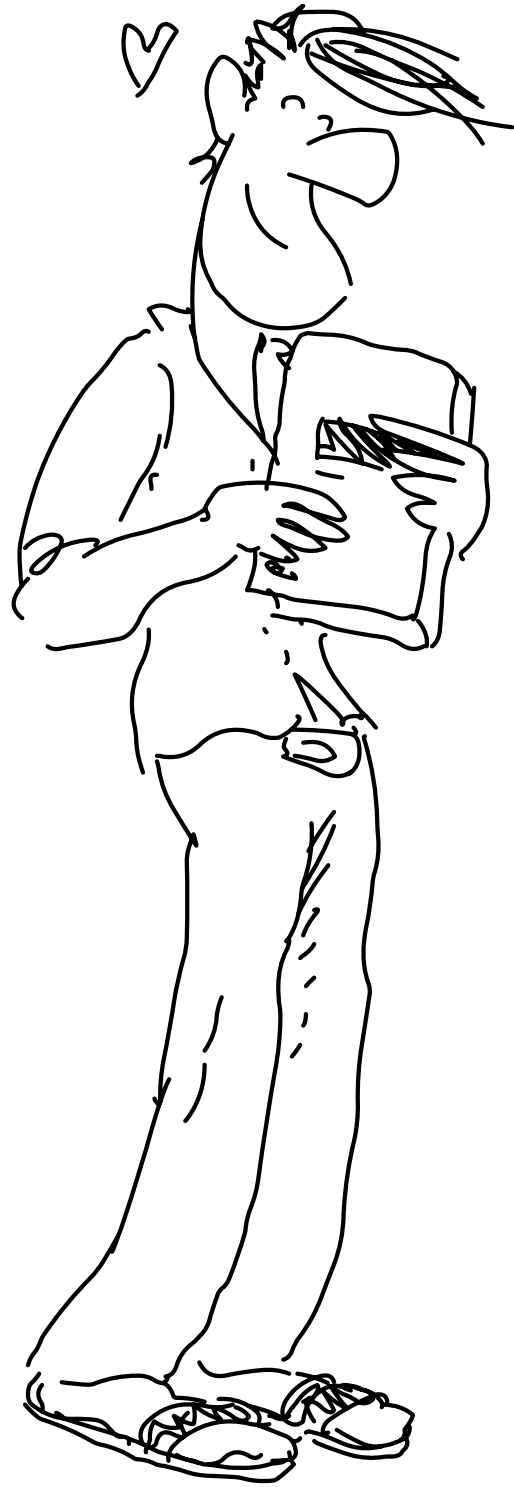
-  »Das ist Schrödinger und seine Lernumgebung«
- »Entwicklungsumgebungen für C++«
- »Erste Schritte in C++«
- »Die C++-Basistypen«
  
-  Inhaltsverzeichnis
-  Index
-  Der Autor
-  Leseprobe weiterempfehlen

Dieter Bär

### Schrödinger programmiert C++ – Das etwas andere Fachbuch

696 Seiten, broschiert, in Farbe, 2. Auflage 2015  
44,90 Euro, ISBN 978-3-8362-3824-3

 [www.rheinwerk-verlag.de/3892](http://www.rheinwerk-verlag.de/3892)



MIT TALENT, KATZEN-  
PHOBIE UND LÄSSIGEM  
SCHUHWERK BESTACH  
SCHRÖDINGER DIE  
RHEINWERK-JURY.  
FÜR IHN GEHT JETZT  
EIN TRAUM IN ERFÜLLUNG.

Was wir Ihnen noch sagen wollten ...

Liebe Leser(in)

C++ ist gnadenlos. Es verzeiht keine Fehler.

# Keinen einzigen.

Aber Sie haben es ja so gewollt. Ihre Wahl ist nun einmal auf diese pingelige Sprache gefallen. Und *wir* können mal wieder die Kohlen aus dem Feuer holen. Warum? Na, wie wir Sie kennen, wollen Sie ja nicht nur C++ lernen. **Sie möchten mehr.** Sie möchten Freude am Lernen haben und Spaß an der Sache: den ultimativen Südbalkon eben.

Genau!  
Lernen ohne  
Leiden! Die Rum-  
kugel essen UND  
sie behalten!

Unsere Antwort darauf lautet:

**Nein, da machen wir nicht mit.** Sie verlangen einfach zu viel. Ein astreines C++-Fachbuch könnten wir bieten. Aber den Spaß an der Sache, den müssen Sie schon selbst mitbringen.  
**Irgendwann ist Schluss.**

Na klar: „Bring gute Laune mit!“ Warum nicht gleich: „Sei doch mal spontan!“?  
Pffft...! Leute!!!

O.K., O.K. Sie sehen ja, wir haben es doch gemacht.

**Das andere Fachbuch.** Mit seriösem ANSI-C++ plus Spaß: Wir zeigen Ihnen die Wege mit der besten Aussicht. Wir bringen Beispiele, die Sie nicht vergessen. Wir gießen dicke Farbeimer über Code. Wir machen uns zum Affen. Und das tun wir nur, weil *Sie* unterhalten werden wollen.

**Endlich zufrieden?**

Bitte sehr. Na, dann viel Spaß,  
**lieber Leser!**

Der Verlag

Moment mal, ich höre dauernd „wir“? Ich, Schrödinger, übernehme doch die Aufgabe (wobei, das mit dem Affen könnt Ihr Euch gleich wieder abschminken). Wäre außerdem klasse, wenn ich dann mal anfangen könnte ...

# Schrödingers Büro

Die nötige Theorie,  
viele Hinweise und Tipps



# Schrödingers Werkstatt

Unmengen von Code, der ergänzt, verbessert und repariert werden will



Fehler/Müll



Schwierige Aufgabe



erledigt



Achtung/Vorsicht



Einfache Aufgabe



Notiz

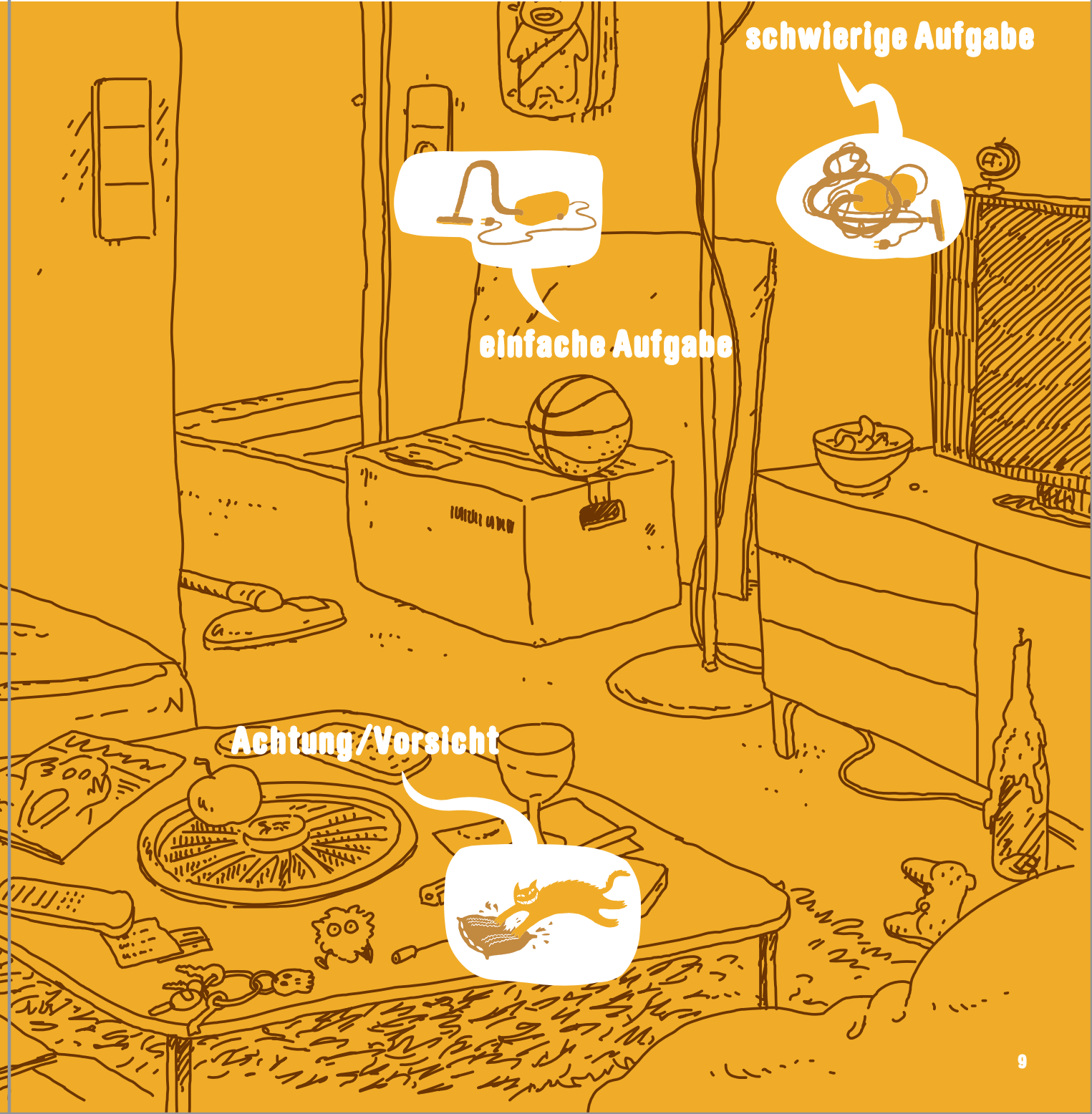


CODE bearbeiten/nachbessern

# Schrödingers Wohnzimmer



Mit viel Kaffee, Übungen und den verdienten Pausen



—EINS—

Entwicklungs-  
umgebungen  
für C++

# Wir richten uns ein ...

**Schrödinger findet heraus, dass er zur Entwicklung von C++-Programmen einen Compiler benötigt. Seine künftigen Kollegen haben ihm gesagt, dass er hierzu zunächst verwenden kann, wozu er Lust hat.**

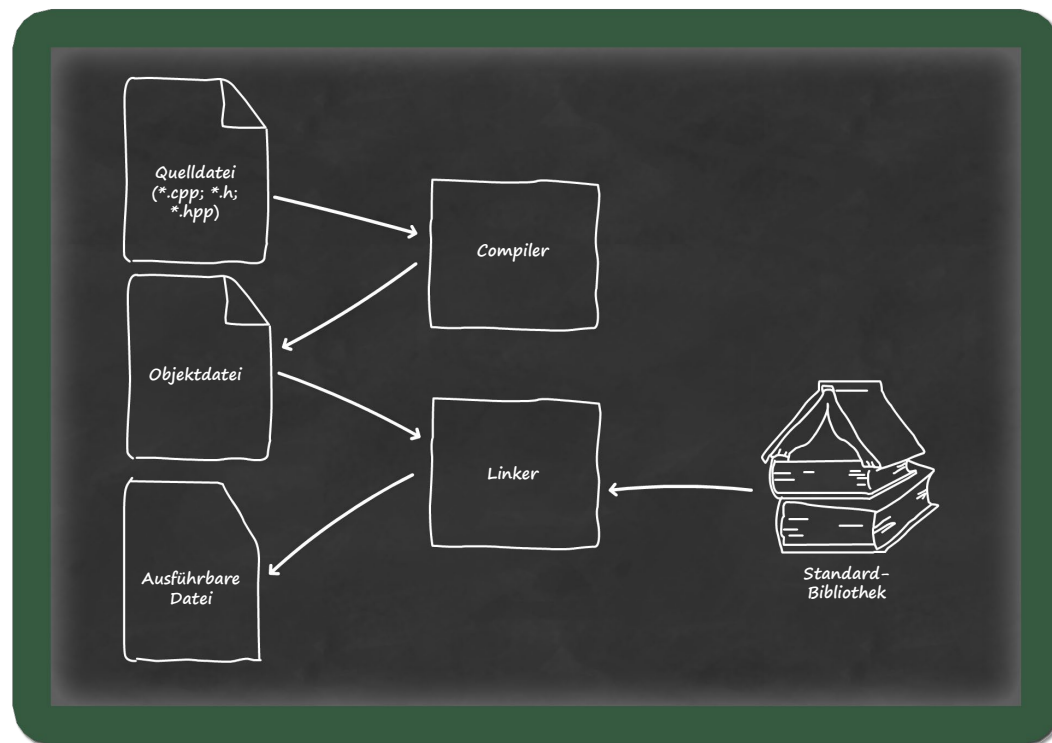
**Keine leichte Aufgabe für Schrödinger, sich im Dschungel von Compilern mit und ohne grafischer Oberfläche zurechtzufinden. Schrödinger hat sich natürlich vorbereitet und keine Kosten und Mühen gescheut, sich seinen Arbeitsplatz einzurichten. Er hat sich gleich drei Rechner zugelegt, um auf Nummer sicher zu gehen und alle Systeme abzudecken: einen Rechner mit Windows, eine Maschine mit Linux und natürlich einen Mac.**

# Brauche ich eine IDE zum Programmieren?

Okay, bevor du überhaupt anfangen kannst, deine Programme zu schreiben, brauchst du natürlich ein Werkzeug, um einen lesbaren Quellcode in einen nicht mehr lesbaren Maschinencode zu übersetzen. Für solche Zwecke benötigst du in C++ einen **Compiler**. Solange du den C++-Standard verwendest, ist dein Quellcode auf die gängigsten Betriebssysteme portierbar. Wenn du den lesbaren Quellcode aber in einen Maschinencode übersetzt hast, dann kannst du diesen nur noch auf dem entsprechenden Betriebssystem ausführen. Damit sollte dir klar sein, dass der übersetzte Maschinencode entweder auf deinem **Windows-Rechner** oder auf dem **Linux-System** oder auf dem **Mac** läuft und nicht mehr portabel ist wie der Quellcode.

### [Hintergrundinfo]

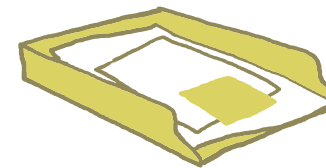
Genau genommen wird der Quellcode von einem Compiler in eine Objektdatei (\*.obj oder \*.o) übersetzt. Diese Objektdatei(en) wiederum wird/werden dann von einem Linker zu einer ausführbaren Datei gebunden. Aber häufig ist bei der Rede von einem Compiler auch gleichzeitig der Linker als komplette Einheit mit gemeint. Aber du solltest trotzdem wissen, dass es sich hierbei um zwei verschiedene Dinge handelt.



Vereinfachte Darstellung, wie man aus einer Quelldatei eine ausführbare Datei macht

Um jetzt aus deinem Quellcode ein echtes Programm (eine ausführbare Datei) zu machen, brauchst du im Grunde nur einen **ASCII-Texteditor**, in den du deinen Quelltext eintippst, und dann eben einen **Compiler** (mit Linker) zum Übersetzen des Quelltextes, um daraus ein Programm zu machen.

Natürlich kannst du hierbei auch ein *Alles-drin-Komplettpaket* mit einer **Entwicklungsumgebung** verwenden. Entwicklungsumgebungen haben den Vorteil, dass eben alles gleich an Bord ist. Hier findest du den Editor, Compiler, Linker und noch viele weitere Dinge wie Debugger, Projektverwaltung, Profiler und noch einiges mehr vor. Der einzige Nachteil von solchen Alles-drin-Komplett-sachen ist halt, dass hierfür ein wenig mehr Einarbeitungszeit nötig wird.

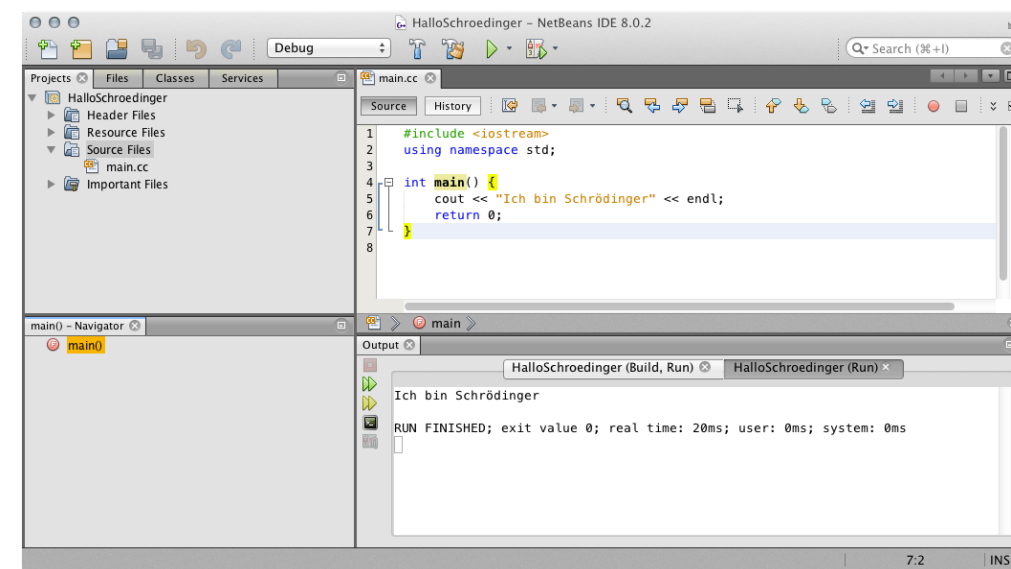


### [Ablage]

Wie dem auch sei, ich bin hier nicht dafür da, für bestimmte Compilerhersteller zu werben, sondern ich will dir lediglich einen kleinen Überblick zu diesem Markt verschaffen. Letztendlich sind alle nur ein Mittel zum Zweck mit demselben Ziel, nämlich ein ausführbares Programm zu erzeugen.

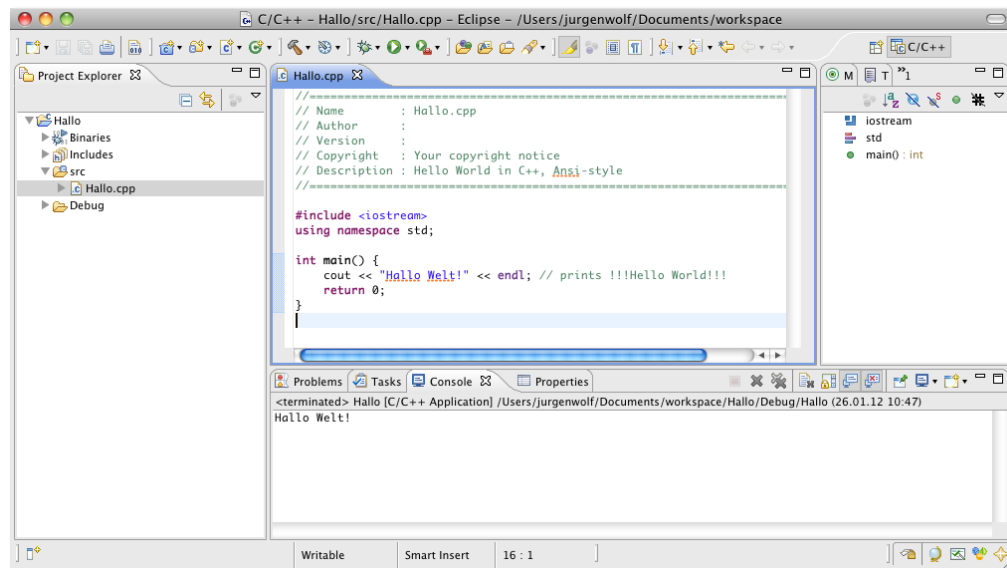
## Multikulturelle Sachen

Es gibt komplette Entwicklungsumgebungen, die für alle gängigen Systeme erhältlich sind und unter Windows, Linux (und Unix-like) sowie Mac OS laufen. Zwei ganz große und umfangreiche Projekte sind die IDEs **Eclipse** und **NetBeans**. Beide Entwicklungsumgebungen wurden in und für Java geschrieben. Aber trotzdem bieten diese auch eine hervorragende Unterstützung anderer Programmiersprachen an, wie u. a. natürlich auch für C++.



Die Entwicklungsumgebung NetBeans für C++



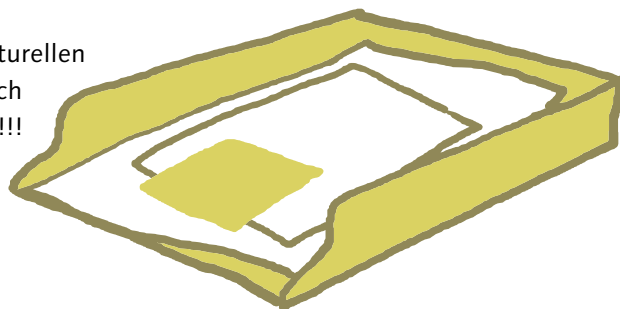


Auch Eclipse bietet eine tolle C++-Unterstützung an.

Ebenfalls für fast alle Systeme vorhanden ist das **Qt SDK** (worin die IDE **Qt Creator IDE** enthalten ist), womit sich neben tollen Anwendungen mit grafischer Oberfläche natürlich auch einfache C++-Programme erstellen lassen. Auch eine sehr interessante Oberfläche stellt **Code::Blocks** zur Verfügung, die mittlerweile auch auf Windows, Linux und Mac OS erhältlich ist.

[Ablage]

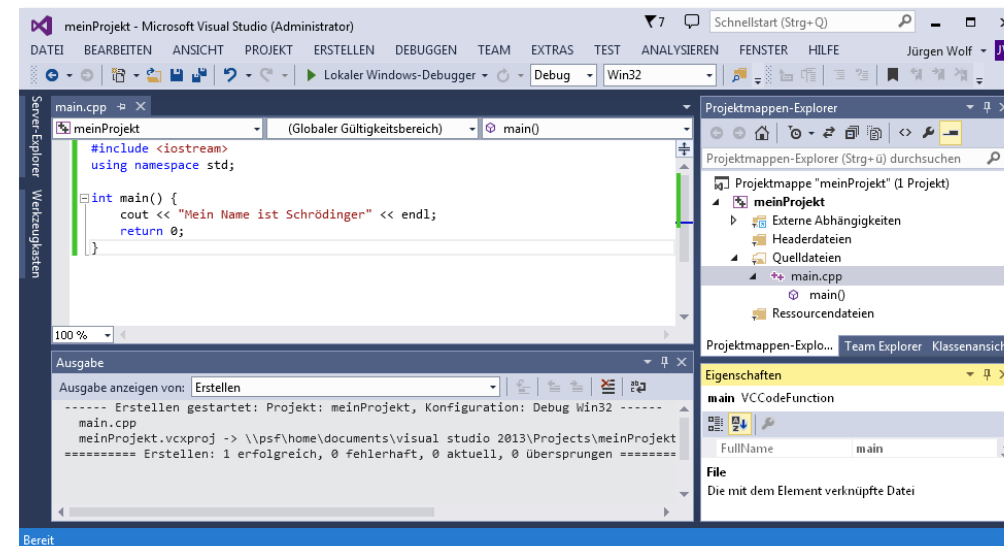
Alle hier erwähnten multikulturellen Sachen sind umsonst erhältlich und kosten dich keinen Cent!!!



## Mikroweiche Sachen

Dominierend auf den Windows-Rechnern dürften wohl die haus-eigenen Produkte sein. Microsoft bietet hierbei unter der Haus-marke **Visual C++ Studio** mehrere kommerzielle Versionen an. Aber es gibt mit **Visual C++ Express** auch eine kostenlose Version aus dem Hause dieser Entwicklungsumgebung, welche dir für den normalen Einstieg oder Umstieg in die C++-Welt vorerst völlig ausreichen dürfte.

Microsoft Visual C++ im Einsatz



Auch sehr beliebt ist der Kommandozeilen-Compiler **MinGW-Compiler**, der eine Portierung des GCC-Compilers auf Windows ist. Darauf aufbauend werden viele andere Entwicklungsumgebungen verwendet. Früher war auch Borlands C++Builder sehr beliebt, der allerdings jetzt nur noch **C++Builder** heißt, weil er nicht mehr Borland gehört.

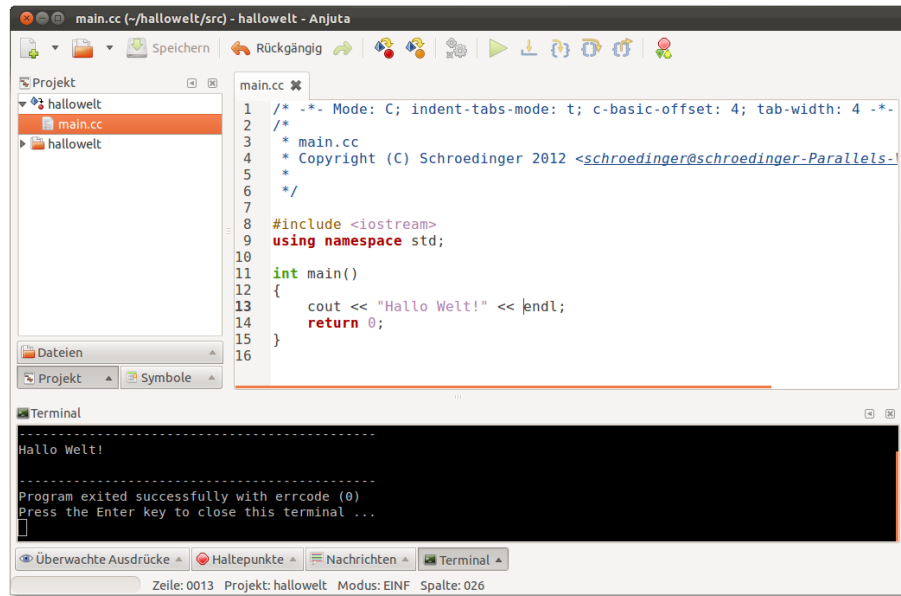
## Die X-Sachen

Auf Linux oder anderen un(ix)artigen Systemen kommt im Grunde fast immer das **GCC-Paket** als Compiler zum Einsatz. Darauf greifen natürlich die meisten IDEs zu. Beliebte Entwicklungsumgebungen sind hier z. B. **KDevelop** und **Anjuta**. Es gibt aber mittlerweile auch viele kleinere interessante IDEs.



[Hintergrundinfo]

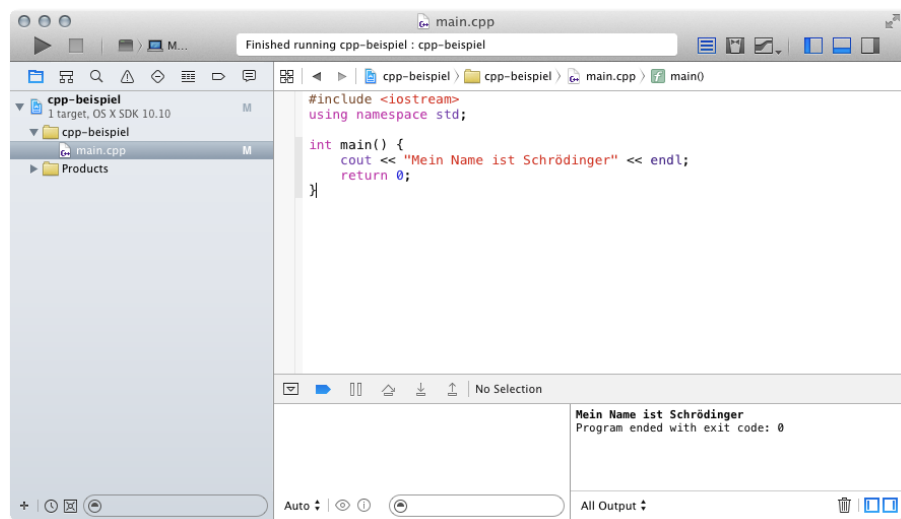
GCC, gcc und g++! Alles dasselbe? Nein, nicht ganz. GCC ist die GNU Compiler Collection (also eine Sammlung von Compilern). gcc (kleingeschrieben) ist der C-Compiler der Sammlung, und g++ ist (ja, richtig) der C++-Compiler von GCC!



Anjuta ist eine sehr angenehme Entwicklungsumgebung (hier unter Ubuntu Linux).

## Angebissene Äpfel

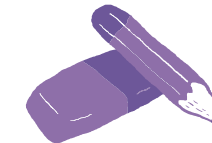
Auch beim Apfel setzt man nach wie vor auf **GCC**. Aber hier sind erste Anzeichen eines Umbruchs zu erkennen. Apple hat schon einige Euros für **Clang** lockergemacht. Clang ist ein Compiler-Frontend für C, C++, Objective-C und -C++. So darf man davon ausgehen, dass Clang über kurz oder lang den GCC auf dem Mac ersetzen wird. Die wohl am meisten eingesetzte Entwicklungsumgebung auf dem Mac OS dürfte ganz klar das hauseigene Produkt **Xcode** sein.



Besonders beliebt auf dem Mac ist Xcode.

## Lass uns endlich loslegen ...

Ja, es gibt wirklich eine gewaltige Menge an (kostenlosen) Compilern bzw. Entwicklungsumgebungen, und du hast die **Qual der Wahl** dabei. Welchen Compiler oder welche Entwicklungsumgebung du hierbei verwendest, hängt natürlich zum einen vom System und zum anderen vom persönlichen Geschmack ab. Da du ja gleich für alle drei Systeme vorbereitet bist, hast du noch mehr Auswahl zur Verfügung.



[Notiz]

Du musst nicht zwangsläufig wie Schrödinger lauter neue Rechner anschaffen, um deinen Code oder deine Beispiele eventuell auf anderen Betriebssystemen oder Compilern zu testen. Hierzu reicht häufig auch eine **Virtualisierung** aus. So kannst du bspw. auf dem Mac-Rechner über eine Virtualisierungssoftware Windows oder Linux „installieren“. Persönlich habe ich gute Erfahrungen mit Parallels Desktop gemacht. Auf Windows oder Linux wiederum verwende ich gerne VMWare, um ein anderes Betriebssystem als Gast zu haben. Ein Kollege von mir (unter Ubuntu Linux) schwört wiederum total auf das kostenlose „VirtualBox SE“ (OpenSource), was auch für Windows erhältlich ist.

*Hättest du das nicht gleich sagen können!  
Dann hätte ich eine Menge Geld gespart!*

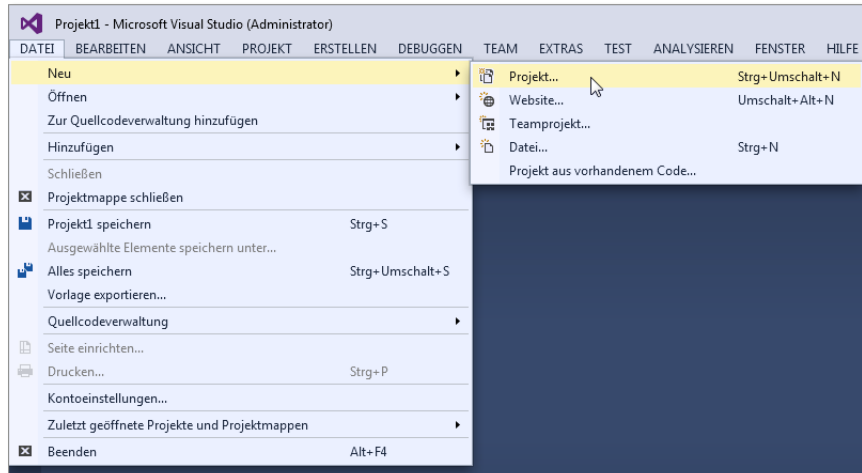


## Übersetzen mit einer Entwicklungsumgebung

Hier möchte ich dir nun eine kleine Anleitung schreiben, wie du aus einem einfachen Quelltext ein ausführbares Programm mit einer Entwicklungsumgebung erstellen kannst. Im Beispiel wird davon ausgegangen, dass du die Entwicklungsumgebung bereits heruntergeladen und installiert hast. Die Anleitung ist natürlich sehr vereinfacht, und hier gilt, dass du dich schon

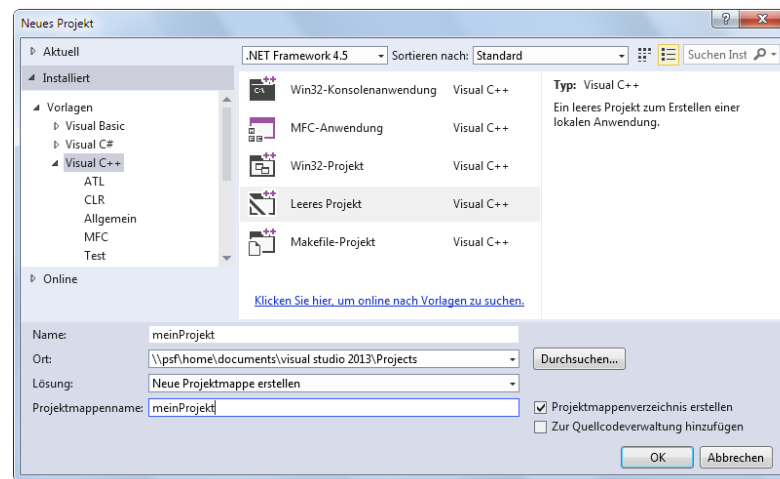
selbst ein wenig mit deiner Entwicklungsumgebung befassen musst. In der Abbildung wurde zwar Microsoft Visual Studio Community dafür verwendet, aber der Vorgang ist bei den meisten anderen Entwicklungsumgebungen auch sehr ähnlich aufgebaut. Nur dass hier und da der Befehl etwas anders lautet und sich woanders befindet.

1. Am einfachsten dürfte es zunächst immer sein, wenn du ein neues Projekt startest. Hier eben über **Datei \* Neu \* Projekt**.



Erster Schritt dürfte immer sein, ein neues Projekt zu starten.

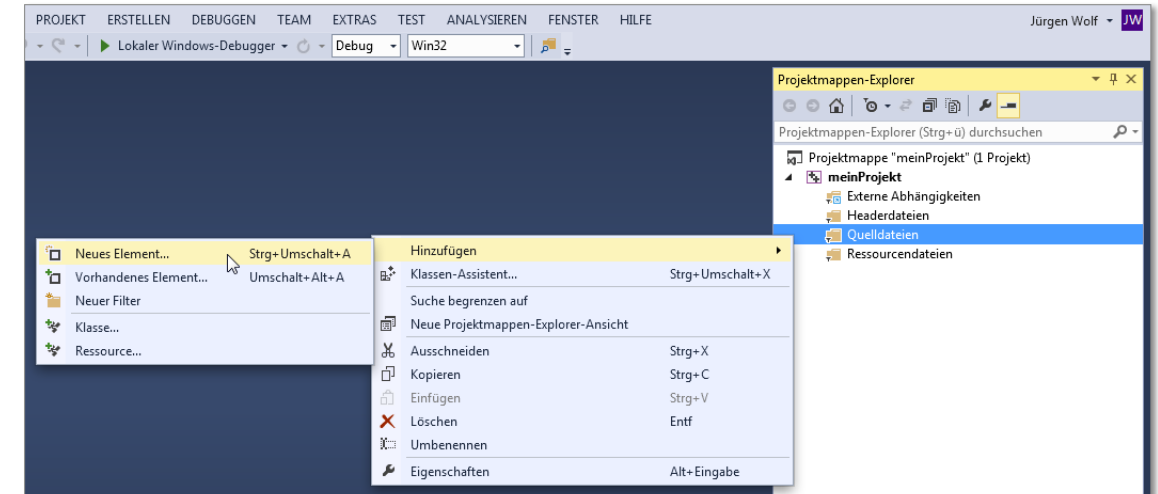
2. Als Nächstes erscheint meistens ein Dialog, der dir dabei hilft, zu entscheiden, was für eine Art von Projekt du erstellen willst, wie du dein Projekt nennen und wo du es abspeichern willst. Hier ist es oft gut, ein leeres Projekt oder eine C++-Konsolenanwendung zu erstellen.



Gewöhnlich hilft dir jetzt ein Wizard weiter, die Art deines Projektes auszuwählen.

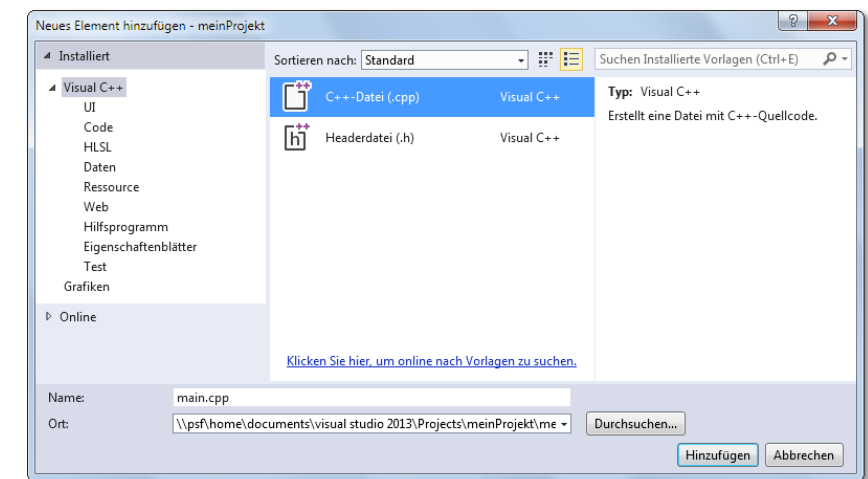
3. Abhängig von der Entwicklungsumgebung findest du jetzt häufig schon einen vordefinierten leeren Quellcode mit seinem Grundgerüst vor. Bei anderen Entwicklungsumgebungen musst du erst noch ein neues Element/eine neue Datei hinzufügen. Neue Elemente wirst du so oder so irgendwann hinzufügen müssen, wenn du deinen Quellcode in mehrere Quell- oder Headerdateien aufteilst. Achte darauf, dass du den Quellcode auch zum Projekt hinzufügst und nicht nur einfach eine neue Datei anlegst!

Neue Elemente/Dateien zum Projekt hinzufügen.

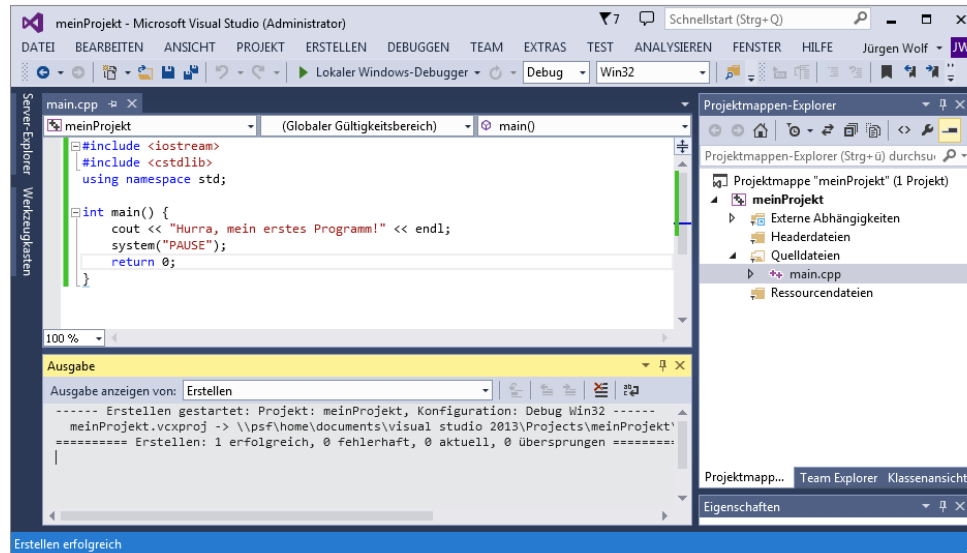


4. Auch hier hilft dir häufig ein weiterer Dialog bei der Wahl, was für eine Art von Datei du zum Projekt hinzufügen willst. Hier ist es eine Quelldatei mit der Endung **\*.cpp**. Aber bei späteren Projekten wirst du auch des Öfteren mal eine Headerdatei mit der Endung **\*.h** hinzufügen.

Meistens gibt es auch einen Wizard, der dir hilft, eine passende Datei hinzuzufügen.



5. Hast du deine Datei(en) hinzugefügt, kannst du anfangen, deinen Quelltext einzutippen. Nach dem Abspeichern kannst du dann den Quellcode übersetzen (kompilieren) und anschließend ausführen. Beim VC++ kannst du den Quelltext über **Erstellen \* Projektmappe erstellen** und **Erstellen \* Kompilieren** übersetzen und dann mit **Debuggen \* Starten ohne Debugging** ausführen.



Jetzt nur den Quelltext eintippen, übersetzen und dann das Programm starten.

**[Fehler]**

Wenn du beim Einstieg hier verzweifelst und wirklich nicht klarkommst, kannst du ja eine E-Mail an Schrödinger senden, der da ziemlich gut durchblickt:

[schroedinger@dieter-baer.de](mailto:schroedinger@dieter-baer.de)

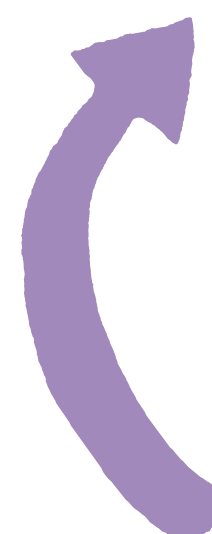
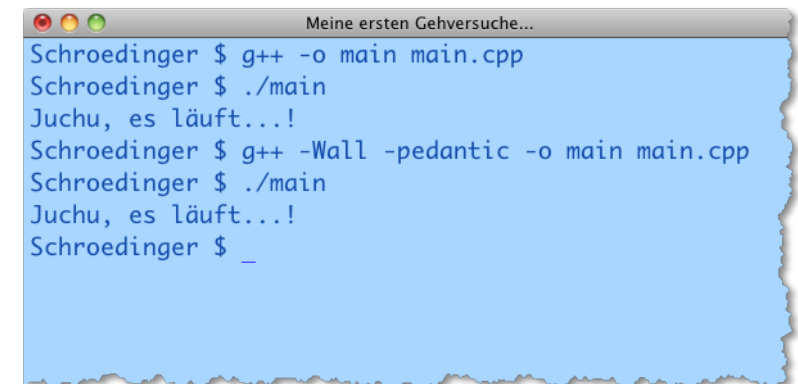
## g++ und clang++

Die Verwendung von Kommandozeilen-Compilern wie **g++** oder **clang++** erscheint dir vielleicht zunächst etwas altbacken, aber hat natürlich den Vorteil, dass du dich nicht mit einem Monster von Programm mit unzähligen Funktionen herumschlagen musst. Im Grunde musst du lediglich deinen Quelltext mit einem **beliebigen ASCII-Editor** schreiben, abspeichern und diesen dann mit einem Kommando übersetzen. Gerade für „kleinere Projekte“ oder Listings im Buch reicht dies völlig aus. Zudem kommt hinzu, dass bei den meisten Linux- und Unix-Systemen der GCC (und somit g++) von Haus aus bereits installiert ist oder gegebenenfalls ganz schnell nachinstalliert werden kann.



**[Notiz]**

Natürlich gehe ich auch davon aus, dass du weißt, was eine Kommandozeile ist und wie du dich mit unterschiedlichen Kommandos dort durch die Verzeichnisse handelst. Schließlich willst du ja Programmierer lernen und hier keine Einführung haben, wie du deinen Rechner bedienen kannst.



Wenn du deinen Quelltext geschrieben und gespeichert hast, brauchst du nur noch in das Verzeichnis zu wechseln, in dem du den Quelltext gespeichert hast, den Compiler anzuwerfen und deinen Code übersetzen zu lassen.

Hier habe ich beim ersten Übersetzungsvorgang den Schalter `-o` verwendet, so dass der Compiler aus der Quelldatei `main.cpp` die ausführbare Datei `main` macht. Du kannst natürlich auch einen anderen Programmnamen für `main` verwenden. Beim zweiten Übersetzungsvorgang habe ich ein paar Schalter mehr verwendet, die mir mehr Informationen über diverse Warnungen zurückgeben, die sonst nicht angezeigt würden.

`-Wall` gibt z. B. sinnvolle Warnungen vom Compiler aus und `-pedantic` gibt Warnungen aus, die vom ANSI-C++-Standard gefordert werden. Willst du deinen Quellcode „nur“ kompilieren, also eine Objektdatei daraus machen, brauchst du nur den Schalter `-c` zu verwenden.



[Notiz]

Es gibt natürlich eine gewaltige Anzahl weiterer Schalter, die du hierbei verwenden kannst. Aber auch hierzu empfehle ich dir, dich bei Bedarf selber ein wenig einzulesen. Vergiss nicht, dass du hier ein C++-Buch vor dir hast, kein Buch über die Verwendung von speziellen Compilern!

## ... am Ende läuft es

So, an der Auswahl von Compilern bzw. Entwicklungsumgebungen mangelt es ja nun wirklich nicht. Daher will ich dir hier noch ein paar Compiler bzw. Entwicklungsumgebungen auflisten, damit du einen Überblick hast. Du kannst diese gerne testen und selber sehen, was dir persönlich am meisten zusagt.

Compiler	Kostenlos	Link	System	IDE
GCC	ja	<a href="http://gcc.gnu.org/">http://gcc.gnu.org/</a>	Win, Linux, Mac, Unix-like	nein
Clang	ja	<a href="http://clang.llvm.org/">http://clang.llvm.org/</a>	Unix-like, Mac	nein
Microsoft Visual Studio	nein/ja	<a href="http://www.visualstudio.com/de-de/downloads/download-visual-studio-vs.aspx">http://www.visualstudio.com/de-de/downloads/download-visual-studio-vs.aspx</a>	Windows	ja
Microsoft Visual Studio Community	ja	<a href="http://www.visualstudio.com/de-de/downloads/download-visual-studio-vs.aspx">http://www.visualstudio.com/de-de/downloads/download-visual-studio-vs.aspx</a>	Windows	ja
Microsoft Visual Studio Express	ja	<a href="http://www.visualstudio.com/de-de/downloads/download-visual-studio-vs.aspx">http://www.visualstudio.com/de-de/downloads/download-visual-studio-vs.aspx</a>	Windows	ja
NetBeans	ja	<a href="http://netbeans.org/">http://netbeans.org/</a>	plattformunabhängig	ja
Eclipse	ja	<a href="http://www.eclipse.org/">http://www.eclipse.org/</a>	plattformunabhängig	ja
Qt Creator IDE	ja	<a href="http://www.qt.io/download/">http://www.qt.io/download/</a>	Win, Linux, Mac	ja
Code::Blocks	ja	<a href="http://www.codeblocks.org/">http://www.codeblocks.org/</a>	Win, Linux, Mac	ja
C++Builder XE2	nein	<a href="http://www.embarcadero.com/products/cbuilder">http://www.embarcadero.com/products/cbuilder</a>	Win, Mac	ja
KDevelop	ja	<a href="http://kdevelop.org/">http://kdevelop.org/</a>	Linux/Unix-like, Mac, Win	ja
Anjuta	ja	<a href="http://www.anjuta.org/">http://www.anjuta.org/</a>	Linux, BSD	ja
Xcode	nein	<a href="http://developer.apple.com/technologies/tools/">http://developer.apple.com/technologies/tools/</a>	Mac OS X	ja
MinGW	ja	<a href="http://www.mingw.org/">http://www.mingw.org/</a>	Windows	nein
Orwell Dev-C++	ja	<a href="http://sourceforge.net/projects/orwelldevcpp/">http://sourceforge.net/projects/orwelldevcpp/</a>	Windows	ja
Intel-C++	nein	<a href="http://software.intel.com/en-us/articles/intel-compilers/">http://software.intel.com/en-us/articles/intel-compilers/</a>	Linux, Win, Mac	nein

Übersicht der Compiler bzw. Entwicklungsumgebungen für C++



Zu den hier erwähnten Compilern muss ich natürlich noch anmerken, dass viele Entwicklungsumgebungen die auf dem Betriebssystem vorhandenen Compiler verwenden und selten eigene Compiler mitliefern. Im Grunde sind ja die Entwicklungsumgebungen nichts anderes als eine grafische Steuerung für deinen Compiler und andere Werkzeuge. So verwenden bspw. unter Windows bekannte Entwicklungsumgebungen wie NetBeans, Eclipse, Qt Creator, Code::Blocks, Dev-C++ oder das MinGW Developer Studio alle **MinGW**. Wobei MinGW wieder auch nur eine Portierung der GNU-Werkzeuge GCC ist.

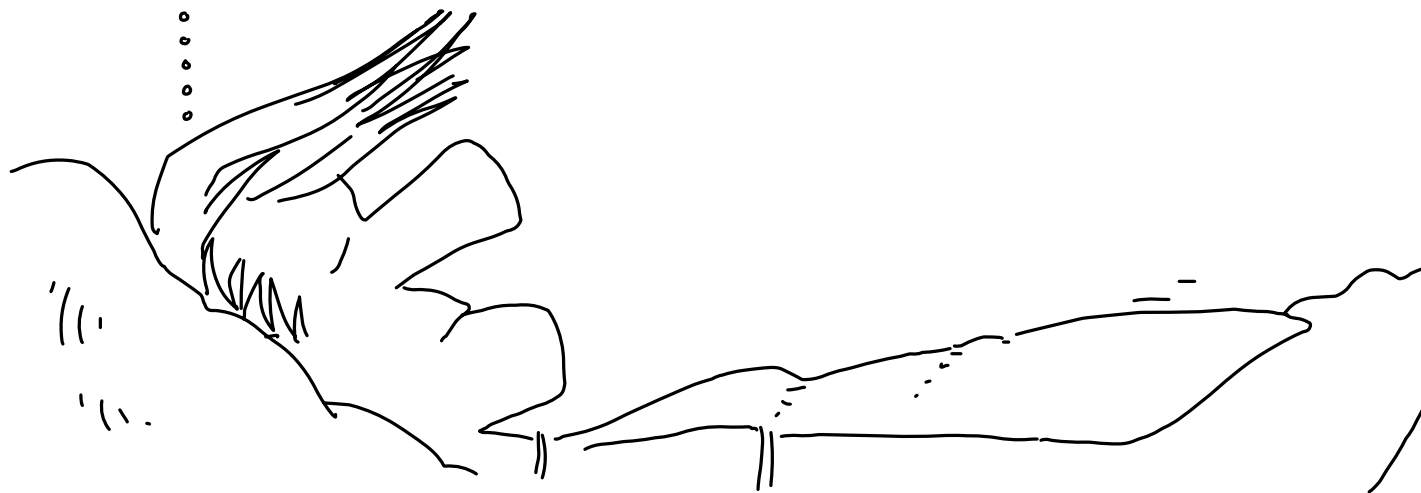
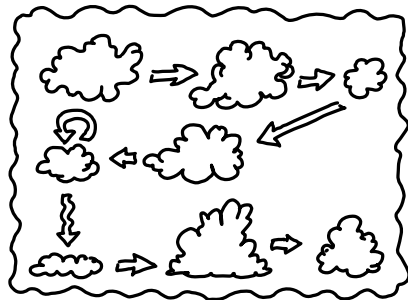
Bei einigen dieser Entwicklungsumgebungen kannst du aber auch den verwendeten Compiler ändern. So kannst du bspw. bei Code::Blocks auch auf den MSVC++-Compiler von Microsoft zurückgreifen, wenn du bspw. MS Visual C++ Express installiert hast. Ähnlich ist es natürlich auch bei Linux-Systemen und Entwicklungsumgebungen wie KDevelop oder Anjuta, welche letztendlich auch wiederum nur auf **GCC** zurückgreifen. Xcode wiederum greift auf das Backend des LLVM-Compilers zurück. Wobei **Clang** das Frontend davon ist.

Somit sind die einzigen **echten Compiler**, die oben aufgelistet sind, der GCC (MinGW), Clang, der MSVC von VC++ und Intel C++ Compiler.



[Notiz]

Unter Linux-Systemen lassen sich einzelne Entwicklungsumgebungen häufig ganz komfortabel mit der Paketverwaltung der entsprechenden Distribution nachinstallieren. Bist du auch ein Fan von Ubuntu, dann kannst du ja mal ein **"sudo apt-get install anjuta g++"** in der Kommandozeile absetzen und zusehen wie alles heruntergeladen und installiert wird.



Erste Schritte in C++

—ZWEI—

# Elefanten können nicht fliegen, aber Schrödinger kann programmieren

Schrödinger schreibt sein erstes Programm und erfährt dabei gleich, was alles zu einem Grundgerüst gehört und wie solche Programme aufgebaut sind.

Schrödinger ist euphorisch, weil alles wie am Schnürchen läuft, und träumt schon von einer hochbezahlten Stelle als Entwickler bei seinem Lieblingsspiel WoW (World of Warcraft). Oder wie wäre es, wenn er nach Steve, Linus und Bill an DEM neuen Betriebssystem arbeiten würde?

# Was ist eigentlich ein Computerprogramm? Ganz kurz und knapp für alle Schrödingers

Ein Computerprogramm liegt gewöhnlich als **ausführbare Programmdatei**, als sogenannter Maschinencode, auf einem Datenträger vor. Startest du ein solches Programm, wird es zunächst in den Arbeitsspeicher des Rechners geladen. Anschließend übernimmt der Prozessor **deines Rechenknechtes** die Kontrolle und verarbeitet der Reihe nach die für ihn lesbare Abfolge von Befehlen – das Programm bei der Ausführung eben.

*Ich habe eben ein ausführbares Computerprogramm mit dem Editor meiner Entwicklungsumgebung geöffnet. Da werden nur komische Zeichen angezeigt!*

Nein, das geht auch nicht mehr mit einem normalen Editor. Bei einem Maschinencode handelt es sich um für den normalbegabten Menschen nicht mehr lesbaren Binärkode, der sich nur noch mit ganz speziellen Maschinensprachmonitoren lesen lässt.

*Verstehe ich nicht! Ich dachte, ich erstelle einen solchen Maschinencode! Wie soll ich denn meine Programme nachträglich wieder ändern?*

**Keine Sorge**, der Maschinencode wird natürlich wieder von einem speziellen Programm erstellt, welches mit einer ganz bestimmten Sprache **gefüttert** werden muss. Diese Sprache ist für dich lesbar und als eine gewöhnliche Textdatei gespeichert, die du jederzeit wieder ändern kannst.

*Mit „Sprache“ meinst du hier C++?*

Richtig! Du schreibst praktisch deine für dich lesbaren C++-Befehle in einen Editor, und dann sorgt ein **Compiler** (und Linker) dafür, dass daraus ein Maschinencode bzw. das ausführbare Computerprogramm erzeugt wird. Wenn du dein Programm jetzt ändern willst, brauchst du nur den **Quellcode** mit deinen C++-Befehlen zu ändern und **erneut zu übersetzen**.

# Die Sache mit dem main-Dings ...



## [Begriffsdefinition]

Irgendwo hat ja alles einen Anfang, also auch ein C++-Programm. Die Mutter aller C++-Anfänge ist hierbei **main()**. Dabei handelt es sich um die Hauptfunktion (main (engl.) = Haupt~), ohne die der Linker niemals ein ausführbares Programm erstellen könnte. Da die **main()**-Funktion also die erste Anlaufstelle deines C++-Programms ist, befinden sich darin auch die ersten Befehle des Programms, welche bei der Arbeit ausgeführt werden sollen.

Hier die nackte **main()**-Funktion, aus der sich tatsächlich auch ein sinnfreies ausführbares Programm erstellen ließe:

Die einzelnen Befehle der **main**-Funktion werden zwischen eine sich öffnende und eine sich schließende geschweifte Klammer gestellt. Die werden als **Anweisungsblock** bezeichnet.

```
int main()
{
    return 0;
}
```

Das ist der erste **Einstiegspunkt** eines jeden C++-Programms, egal wie viele andere Funktionen davor stehen oder danach noch folgen mögen.

Da die Funktion **main()** einen Wert zurückgibt (will das **int** vor **main()** so haben) wird mit dem Befehl **return** der Wert 0 an den Aufrufer des Programms zurückgegeben.

[Zettel]

Genaugenommen musst du bei der Funktion `main()` nicht unbedingt (explizit) den Wert 0 zurückgeben. Tust du das nicht, macht `main()` das (implizit) für dich. Allerdings ist dies nur bei der `main()`-Funktion gültig und somit eine Ausnahmeregelung.

*Zwischen die komischen Klammern kommen also die Befehle für das main-Dings! Welche Befehle überhaupt, und wie kann ich diese voneinander trennen?*

**Schön, dass du mitdenkst!** Die einzelnen Befehle werden mit dem Semikolon-Zeichen am Ende abgeschlossen. Jeder Ausdruck, der mit diesem Zeichen endet, wird als Anweisung bzw. Befehl behandelt. Der Compiler weiß dann, hier ist das Ende des Befehls und arbeitet dann den nächsten Befehl ab.

*Welche Befehle denn jetzt nochmal?*



[Achtung]

Denk daran, dass man in C++ ganz streng zwischen Groß- und Kleinbuchstaben unterscheidet. So kannst du nicht einfach hergehen und `Main()` statt `main()` schreiben. Der englische Fachbegriff hierzu lautet **Case sensitivity**.



## Unser erstes main-Dings soll laufen ...

Jetzt ist es an der Zeit, dass du deinen ersten eigenen Code eintippst und zur Ausführung bringst. Es geht noch gar nicht darum, was welcher Befehl macht, sondern nur, dass ein Fenster aufpoppt und irgendwas vor sich **hinbrabbelt**.

**Hier dein erstes Programm:**

```
#include <iostream> *1
```

```
using namespace std; *2
```

```
int main()
```

```
{
```

```
    cout << "Elefanten können nicht fliegen,\n"; *3
```

```
    cout << "aber Schrödinger kann programmieren\n"; *3
```

```
    return 0;
```

```
}
```

\*1 Das ist ein Befehl

für den **Präprozessor**. Der Präprozessor ist ein weiteres Programm, welches noch vor dem Compiler ausgeführt wird. Damit sagst du dem Compiler, dass dieser Befehle verwenden kann, welche in der **Headerdatei** bzw. auch in der Bibliothek **iostream** enthalten sind.

\*2 Damit

stellst du für dein Programm den **Namensraum std** zur Verfügung. **Das ist praktisch**, weil du ohne diese Zeile Befehle aus diesem Namensraum erst mit **std::cout** oder **std::endl** qualifizieren müsstest.

\*3 Das sind deine ersten echten Befehle für das Programm. Über **cout** und den Operator **<<** wird der Text, der **zwischen den Gänsefüßchen** steht, auf dem Bildschirm ausgegeben. Hättest du **iostream** am Anfang des Listings nicht angegeben, würde sich das Programm nicht übersetzen lassen, weil der Compiler den Befehl dann nicht kennt.

*Juhu, es läuft ...*



```
Terminal — bash — 50x15
Dieter Baer $ ./main
Elefanten können nicht fliegen,
aber Schrödinger kann programmieren
Dieter Baer $
```

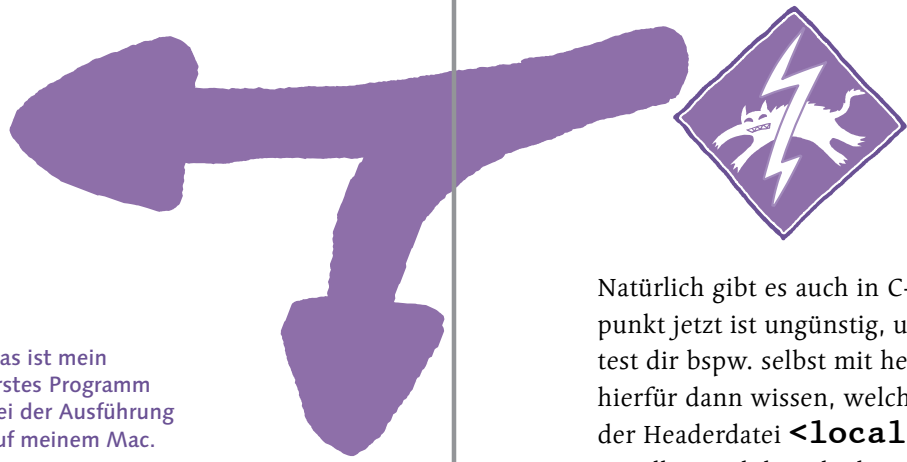
Das ist mein  
erstes Programm  
bei der Ausführung  
auf meinem Mac.

```
user@ubuntu:~
Datei Bearbeiten Ansicht Suchen Terminal Hilfe
user@ubuntu:~$ ./programm001
Elefanten können nicht fliegen,
aber Schrödinger kann programmieren
user@ubuntu:~$
```

Heureka,  
mein Programm  
auf meinem  
Ubuntu-Linux

```
C:\Qt\2010.05\qt\programm001-build-desktop\debug\pro...
Elefanten können nicht fliegen,
aber Schrödinger kann programmieren
Drücken Sie eine beliebige Taste . . . _
```

Und zu guter Letzt  
das ausführende  
Programm  
unter MS Windows



[Achtung] Wenn falsche Umlaute in der MS Windows-Box angezeigt werden, hat das historische Gründe. In der klassischen DOS-Eingabeaufforderung wurde zur Kodierung des Textes die **Codepage 850** (Dos-Latin.1) verwendet. In der grafischen Oberfläche von Windows wird die Eingabeaufforderung hingegen mit Codepage 1252 (Win-Latin-1) kodiert.

Natürlich gibt es auch in C++ einige Mittel, um Umlaute deiner Kultur darzustellen, aber ich denke, der Zeitpunkt jetzt ist ungünstig, und dann würden hier nur noch mehr Fragezeichen abgedruckt werden. Du könntest dir bspw. selbst mit hexadezimalen Werten (bspw. eine UTF-8-Literale) behelfen, allerdings musst du hierfür dann wissen, welche Codepage eingestellt ist. Auch findest du z. B. eine Klasse **std::locale** in der Headerdatei **<locale>** für solche Zwecke vor. Du musst also „nur“ ein neues Objekt von diesem Typ erstellen und das Objekt zu deinen Kulturkreisen („German“) anpassen. Folgendes Rezept könnte das Problem z. B. beheben:

```
#include <locale> // benötigter Header
#include <iostream>
using namespace std;
int main()
{
    locale loc; // Objekt erstellen
    // mit deiner Kultur initialisieren
    locale::global(locale("German"));
    // Jetzt sollte es ein ö sein
    cout << "Schrödinger" << endl;
    ...
}
```



[Notiz] Hier kannst du dich eines alten Tricks bedienen. Verwende einfach den MS-DOS-Befehl **PAUSE** dafür, welcher die Ausführung des Programms bis zum nächsten Tastendruck anhält. Den Befehl kannst du in C++ mit der (leider) alten C-Funktion **system()** absetzen. Hierzu musst du zusätzlich noch die Headerdatei **<cstdlib>** mit angeben, in der der Befehl enthalten ist.



Und wieder dieses Windows!  
Das Fenster poppt nur kurz auf  
und verschwindet wieder!

Einige Entwicklungsumgebungen kümmern sich selbst darum, bei anderen musst **du dich darum kümmern**. Im Grunde ist es ja kein Fehler, wenn das Programm wieder verschwindet. Letztendlich wird das Programm ja von Anfang bis Ende ausgeführt. Leider findet die Ausgabe hier ja auf die MS-DOS-Eingabeaufforderung statt, die extra für die Ausgabe geöffnet wird und sich daher am Ende auch automatisch wieder schließt, ohne dass einer gesehen hat, was drin stand.

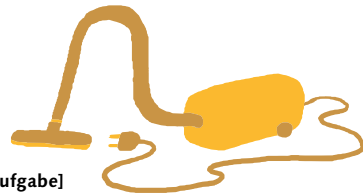
```
#include <iostream>
#include <cstdlib>
int main()
{
    cout << "Elefanten können nicht fliegen,\n";
    cout << "aber Schrödinger kann programmieren\n";
    system("PAUSE");
}
```



Hier dein Rezept nochmals mit diesem Notbehelf für **schnellpoppende Windows-Fenster** auf dem Zettel.

## Endlich entspannen und träumen!

Wenn das weiterhin so einfach ist, könnte ich mich doch bei Blizzard Entertainment für die nächste Auflage von WoW bewerben. Das wäre doch eine Karriere, vom Entwickler für Software von Schuhkartons zum Chef-Entwickler eines Top-Software-Konzerns...



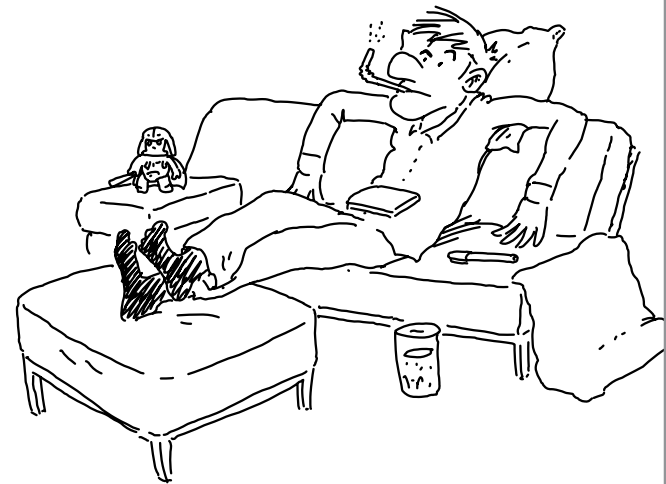
[Einfache Aufgabe]

Bevor du jetzt deine Bewerbung schreibst, kannst du mir sicherlich die Fehler im folgenden Listing zeigen?

```
#include <iostream>

using namespace std;

int Main() { cout << "WoW\n" }
```



[Notiz]

In diesem Beispiel wurde der **erste Buchstabe** von `Main()` **großgeschrieben**. In C++ wird allerdings strengstens zwischen Groß- und Kleinschreibung unterschieden. Außerdem wurde beim Kommando über `cout` das **Semikolon** am Ende des Befehls **vergessen**. Das Weglassen des Rückgabewertes 0 am Ende mit `return` ist in der `main()`-Funktion kein Fehler und erfolgt (nur hier) automatisch, sofern nichts dransteht.

Rückgabe von `main()`

Dich stört vermutlich ein wenig die lasche Haltung in Bezug auf `return` bei `main()`. An dieser Stelle solltest du noch wissen, dass es trotzdem immer `int main()` heißen muss. Der Standard will, dass du hier immer ein `int` vor `main()` schreibst und nichts anderes sonst. **Lass dir von keinem was anderes erzählen**. Das schreibt der Standard eben so, und es gibt da keine Diskussion!

## Kreuz und quer oder alles in Reih und Glied?

Ein paar Worte möchte ich noch zur Formatierung des Quellcodes verlieren, ehe du eine wohlverdiente Pause einlegen kannst. Zunächst einmal kannst du deinen Code eintippen, wie du willst. Im Grunde ist es egal, wie du das formatierst, solange ein **Befehl mit einem Semikolon abgeschlossen** wird und mehrere zu einer Funktion (oder auch einem Ausdruck) gehörende Befehle in einem Anweisungsblock zwischen den geschweiften Klammern zusammengefasst sind. Also folgendes Chaos ist ohne weiteres erlaubt:

```
#include <iostream>

using namespace std;int main()

{cout << "Kreuz und quer\noder alles in Reih"

" und Glied...?! \n";return 0;}
```

Trotzdem lohnt es, sich die Mühe zu machen, den Code etwas ordentlicher zu formatieren. Ein **gleichmäßiges Einrücken** innerhalb eines Anweisungsblocks hat noch niemandem geschadet. Auch ist es hilfreicher, wenn du nur einen Befehl pro Zeile schreibst. Das hilft dir im Falle eines Fehlers; und wenn es nur ein Typo in der Syntax ist, der so aufgrund der Fehlermeldung des Compilers schneller gefunden wird. Außerdem werden es dir deine Mit-Programmierer danken, wenn Sie deinen Code auch mal lesen oder überarbeiten müssen.

## Kein Kommentar?

Du hast zwei Varianten, einen Kommentar in deinem Quellcode zu vermerken:

- ☛ Soll sich dein Kommentar nur über eine Zeile erstrecken, reicht die Zeichenfolge `//`. Alles, was sich jetzt dahinter befindet, wird vom Compiler ignoriert.

```
// Ich bin ein Kommentar
```

- ☛ Mehrzeilige Kommentare leitest du mit `/*` ein und beendest diese mit `*/`. Alles, was sich also zwischen `/*` und `*/` befindet, wird vom Compiler ignoriert:

```
/* Ich
bin ein
Kommentar */
```

[Belohnung]

So, jetzt kannst du dich zurücklehnen und von deiner künftigen Karriere träumen.



## Wie komme ich hier zum Bildschirm ...?

Um jetzt sinnvollere Programme zu erstellen, fehlt dir eine kleine **Wegbeschreibung**, wohin oder woher die Daten kommen, genauer, wie du etwas von der Tastatur einlesen kannst oder wie du die Daten dann auf den Bildschirm bekommst.



### [Hintergrundinfo]

Zunächst einmal ist die Sprache C++ ohne die **iostream**-Bibliothek stumm und taub. Erst diese Bibliothek ermöglicht dir Wege, etwas von der Tastatur einzulesen oder etwas auf dem Bildschirm auszugeben.

Das ist auch der Grund, warum du bei fast jedem Programm die Headerdatei **<iostream>** mit angeben musst. Der Hauptgrund, die Ein- und Ausgabe von der Sprache zu trennen und als Bibliothek anzubieten, macht es wesentlich einfacher für den Compilerhersteller, die Sprache auch auf **anderen Systemen** anzubieten. Daher kannst du deinen Quellcode auf deinem Mac-, Linux- oder aber auch Windows-Rechner übersetzen und auf die gleich Art und Weise verwenden.

### [Zettel]

Das Konzept der **iostream**-Bibliothek ist natürlich weitaus umfangreicher und komplexer, als das jetzt hier beschrieben wird. **Fürs Erste** musst du dich hier erst einmal damit zufriedengeben, dass du diese Bibliothek für die einfache Ein- und Ausgabe verwendest.



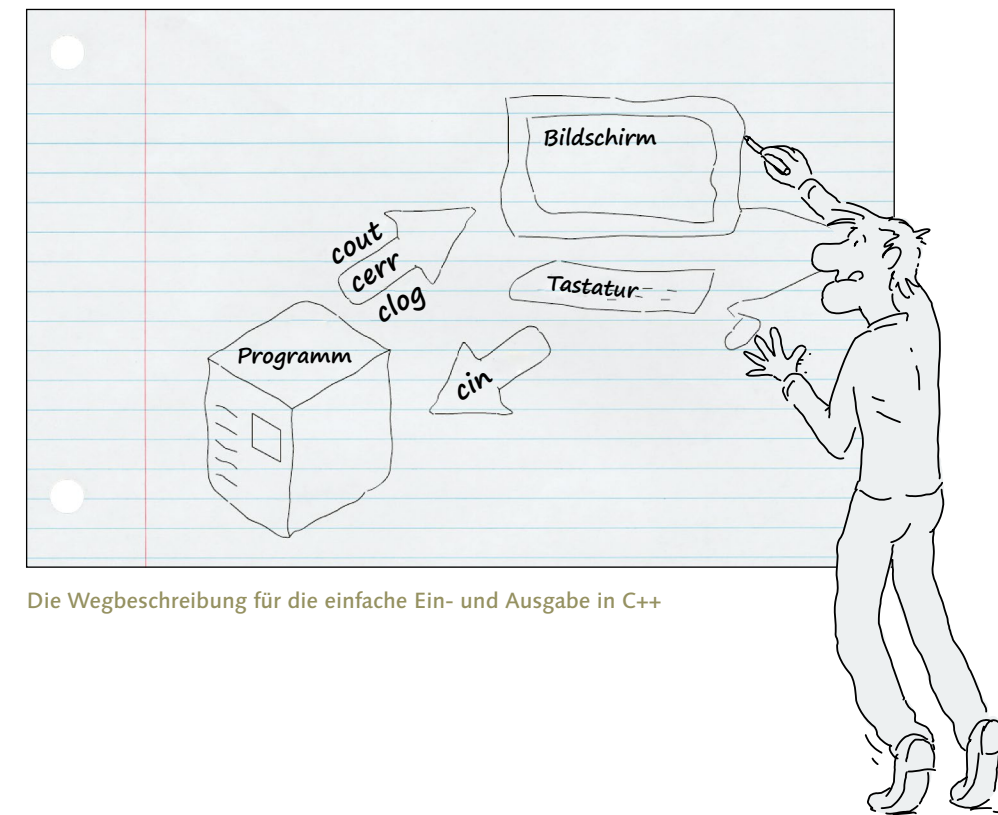
### [Begriffsdefinition]

In C++ wird bei diesem Datenverkehr von einem **Stream** (engl. für [Daten-]Strom) gesprochen. Diesen Stream kannst du dir wie einen Tunnel vorstellen, durch den die einzelnen Bytes mit Daten einfach durchgeschoben werden.

Abhängig davon, aus welcher Richtung die Daten kommen, kann die **Quelle eines Streams von der Tastatur oder einer Datei herrühren**. In der anderen Richtung kann der Zielort dann der Bildschirm oder auch wieder eine Datei sein. Du erzeugst quasi ein solches Stream-Dings, welches fest mit deinem Programm verbunden ist, so dass der Datenverkehr des Programms dann nur noch über diesen Stream erfolgt.

Die grundlegenden Streams für die einfache Ein- und Ausgabe von Tastatur und Bildschirm von der **iostream**-Bibliothek lauten somit:

Objekt	Bedeutung	Wohin/Woher
cout	Standardausgabe	Bildschirm
cerr	Standardfehlerausgabe	Bildschirm
clog	Standardfehlerausgabe (gepuffert)	Bildschirm
cin	Standardeingabe	Tastatur



Die Wegbeschreibung für die einfache Ein- und Ausgabe in C++

## Auf dem Weg zum Bildschirm

\*1 Kannst du zur **üblichen Ausgabe** auf dem Bildschirm verwenden.

Unterwegs in Richtung Bildschirm geht es also mit den Stream-Dingern **cout**, **cerr** und **clog**. Zum Beispiel:

```
cout << "Auf zum Bildschirm\n"; *1
```

```
cerr << "Das auch...\n"; *2
```

```
clog << "Und noch einer...\n"; *3
```

\*2 Ist die Standardausgabe nicht mehr verfügbar oder tritt ein Fehler auf, solltest du diesen Stream verwenden. Im Gegensatz zu **cout** wird **keine Pufferung** für diese Ausgabe verwendet.

\*3 Diesen Stream kannst du verwenden, wenn du **Kontrollmeldungen** auf dem Bildschirm ausgeben willst. Im Gegensatz zum Stream **cerr** wird die Ausgabe allerdings gepuffert.

## Badewanne schon voll?

Sinnvollerweise wird der Datenverkehr mithilfe von **Puffern** geregelt. Stell dir mal vor, es wird jedes einzelne Zeichen sofort über den Stream auf dem Bildschirm ausgegeben oder in eine Datei geschrieben. Wenn jedes Programm auf dem Rechner so vorgehen würde, gäbe es wohl keine Interaktion mehr mit dem Rechner, und der Prozessor wäre bis zum Anschlag beschäftigt. Zudem sind diese Arbeiten außerdem nicht unbedingt die schnellsten Aktionen und würden den Rechner unnötig ausbremsen. Daher gibt es einen Puffer, der **wie eine Badewanne** gefüllt werden kann, bis nichts mehr reinpasst. Erst danach kann man den Stöpsel ziehen, und das Wasser kann wieder ablaufen.

**Natürlich gibt es hier auch Ausnahmen, und man kann den Stöpsel auch schon bei einer halbvollen Badewanne ziehen.**



[Hintergrundinfo]

Der Stream **cerr** arbeitet **ungepuffert**, weshalb die Ausgabe sofort auf dem Bildschirm durchgeführt wird. Bezogen auf die Badewanne entspricht dieses Verhalten dem Einlassen von Wasser in die Wanne, obwohl der Stöpsel nicht abgedichtet wurde. Das Wasser läuft also vom Hahn sofort in den Ausfluss.

## Gib mir fünf

In die andere Richtung geht es mit dem Datenstrom **cin**, welcher in der Regel für das Einlesen von der Tastatur verwendet wird.

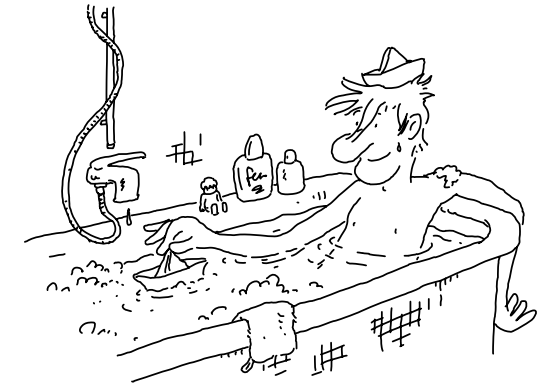
\*1 Das ist **int** (=Integer = ganze Zahl) mit dem Namen **gibmirfuenf**.

```
int gibmirfuenf; *1
```

```
cout << "Gib ihm fünf: ";
```

```
cin >> gibmirfuenf; *2
```

\*2 Hier kannst du **gibmirfuenf** über die Tastatur geben, was er haben will



## Stream me up, Scotty

Damit die Ströme für die Ausgaben **cout**, **cerr** und **clog** bzw. der Eingabe **cin** richtig funktionieren, musst du den Ausgabeoperator **<<** und den Eingabeoperator **>>** verwenden. Beide Operatoren sind so implementiert (**überladen, um genau zu sein**), dass du damit die grundlegenden Basisdatentypen ohne besondere Vorkehrungen ausgeben und einlesen kannst. Natürlich sollte dir dabei klar sein, dass du den Ausgabeoperator nur für Ausgabe-Streams und den Eingabeoperator nur für den Eingabe-Stream verwenden kannst.

## Gegenseitige Wahrnehmung ...

Jetzt ist es an der Zeit, die einseitige Beziehung zwischen dir und deinem Programm zu beenden. Das folgende Beispiel zeigt dir eine gut gepflegte Beziehung zwischen dem Bildschirm und der Tastatur:

```
#include <iostream>

using namespace std;

int main()
{
    int gibmirfuenf;

    cout << "Gib mir fünf: "; *1
    cin >> gibmirfuenf; *2
    cout << "Danke für die " << gibmirfuenf << endl; *3
    return 0;
}
```

\*3 Da der Operator << überladen werden kann, lässt sich dahinter immer noch weiterer Text mit einem weiteren << nachschieben. Das **endl** am Ende ist ein kleines Helferlein (mit dem mächtigen Namen Manipulator) und hüpfert für uns in die nächste Zeile.

\*1 Die Aufforderung für eine Eingabe.

\*2 Hier musst du eine Ganzzahl eingeben. Über den Operator >> wird die Ganzzahl nach **gibmirfuenf** geschoben.



(Fehler)

Der Datenverkehr über das **cin**-Dings von der Tastatur wird an der Stelle beendet, an der das erste Zeichen nicht mehr bearbeitet werden kann. Wenn du bei der Eingabe von **gibmirfuenf** den Wert „5Meter“ eingegeben hast, so wird in **gibmirfuenf** nur der Wert „5“ gespeichert. Das liegt daran, dass du als Typen einen Integer verwendest. Führende Leerräume hingegen werden völlig ignoriert.

## Manipulieren oder selber steuern?

Um bei der Ausgabe in die nächste Zeile zu springen, hast du mehrere Möglichkeiten:

1. Entweder du manipulierst mit **endl**, flüchtest mit **'\n'**.
2. Oder du flüchtest und terminierst mit **"\n"**.
3. Alle drei Zeilenende-Dinger kannst du über den Ausgabeoperator << in einen Ausgabe-Stream stecken.
4. Als vierte Möglichkeit kannst du noch den Ausgabeoperator etwas schonen und das Zeichen für das Zeilenende am Ende des Textes hinzufügen (zum Beispiel: **"Schonendes Zeilenende\n"**).



[Notiz]

Bei den Zeichenkombinationen, die mit dem Zeichen \ beginnen, handelt es sich um nicht darstellbare Steuerzeichen, die auch Escape-Sequenzen (escape = entfliehen, entkommen) genannt werden.

*Hum, das sollte eigentlich nicht so schwer sein. Ich muss doch nur diese Zeilenenden in den Ausgabeoperator << stecken, so dass am Ende alles irgendwie beim Stream **cout** ankommt.*

\*1 Der Manipulator **endl** stellt die langsamste Version da, weil noch eine **Extra-Synchronisation** ausgeführt wird. Und zwar werden hier alle sich noch im Puffer befindlichen Daten sofort an die Ausgabe geschickt.

```
#include <iostream>

using namespace std;

int main()
{
```

```
    cout << "Der Manipulator" << endl; *1
```

```
    cout << "Das Zeichenliteral" << '\n'; *2
```

```
    cout << "Das Stringliteral" << "\n"; *3
```

```
    cout << "Oder gleich im Stringliteral\n"; *4
```

```
    return 0;
```

```
}
```

\*3 Hier wird neben dem Zeilenendezeichen noch ein zusätzliches (aber nicht sichtbares) **Terminierungszeichen** verwendet, welches das Ende des Strings kennzeichnet (Hasta la vista, baby!).

\*2 Die wohl **sparsamste Lösung**, weil nur ein einzelnes Zeichen verwendet wird.

\*4 Die wohl für den Prozessor **günstigste Lösung**, womit du den Ausgabeoperator << nicht nochmals extra belasten musst.



## Noch ein wenig Brain-Streaming

Eigentlich sollte jetzt Zeit für **eine Runde WoW** sein, aber vorher wollen wir den Tag noch ein wenig überfliegen. Du weißt jetzt, dass der Datenverkehr in C++ über Streams realisiert wird. Zwar hat das Stream-Konzept noch viel mehr zu bieten, aber für die nächsten Buchseiten bist du hiermit erst mal **gut gerüstet**. Du weißt jetzt auch, dass diese Streams nicht Bestandteil der Sprache, sondern als Extra-Bibliothek enthalten sind. Des Weiteren hast du auch die Operatoren `<<` und `>>` kennengelernt, die dir dabei helfen, die Daten korrekt in den Stream zu stecken.

[Einfache Aufgabe]

Bevor du dich also auf **neue Quests** in der Welt der Kriegskunst stürzen kannst, möchte ich noch sehen, ob du das Thema verstanden hast. Du solltest jetzt in der Lage sein, im folgenden Listing ohne Lupe und Rechner, die Fehler mit dem **bloßen Auge** zu erkennen.

```
#include <iostream>

int main()
{
    int level;
    cout >> "Welchen Level hast du in WoW: ";
    cin << level;
    cout >> "Wow, Level " >> level >> "! Nicht schlecht\n";
    return 0;
}
```



In dem Beispiel wurde bei dem Stream `cout` statt des Operators `<<` der Eingabeoperator `>>` verwendet und bei `cin` wurde statt des Operators `>>` der Ausgabeoperator `<<` verwendet. Im Beispiel sind also die Richtungen für die Operatoren vertauscht worden.



**Richtig**, aber damit dieses Beispiel auch läuft, fehlt noch etwas ganz Wichtiges.



**Okay, ich helfe dir.**

Wenn du jetzt die Operatoren auf den richtigen Weg bringst, gibt es trotzdem keine Verbindung zu den Streams `cout` oder `cin`, weil diese nun mal keiner kennt. **Wie ein Vampir**, der nicht ins Haus kommt, wenn du ihn nicht extra herein-gebeten hast, musst du auch die beiden Streams in den Gültigkeitsbereich bitten. Es fehlt praktisch der Namensbereich `std` in diesem Beispiel, den du bisher immer mit **using namespace std**; herein-gebeten hast. Alternativ kannst du aber auch den Namensbereich `std` mit dem Bereichsoperator `::` reinbitten:

```
std::cout << "Du darfst rein"
std::endl;
```

**So, jetzt kannst du einen neuen Quest anfangen.**



—DREI—

# Verschiedene Typen für einen bestimmten Zweck

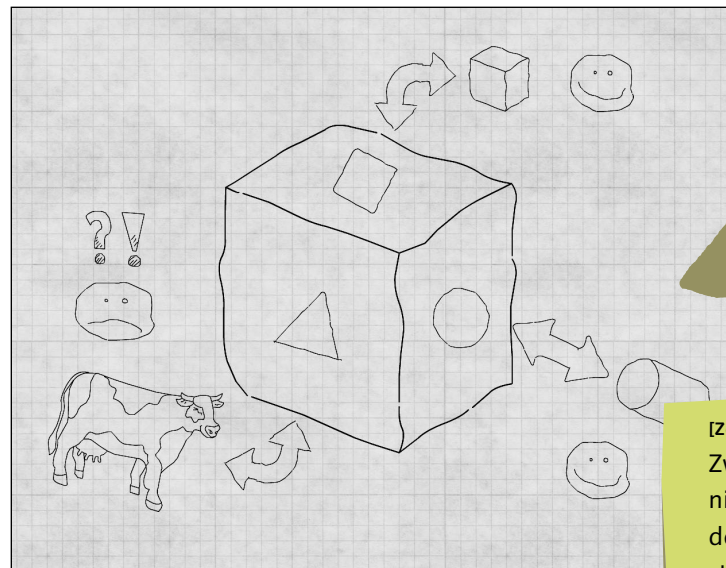
„Wohin mit den Daten?“, fragt sich Schrödinger und bemerkt, dass es dafür verschiedene Basisdatentypen gibt und dass C++ außerdem sehr pingelig ist, was diese Typen betrifft. Hierbei erfährt er auch gleich, wie die einzelnen Zeichen und Buchstaben in seinen Computer gelangt sind und wie er sie da wieder heraus und auf den Bildschirm bekommt.



## Starke Typen

Um **Daten speichern** zu können, benötigst du eine **Kiste**. Statt einer geöffneten Kiste findest du hier allerdings eher Öffnungen mit geometrischen Formen, wie du das noch von Steckspielen aus deiner Kinderzeit her kennen könntest. Das bedeutet also, dass die jeweiligen Formen nur in **bestimmte Öffnungen** passen. So ist das auch mit dem Speichern von Daten.

Man kann nicht alles reinstecken, wo und vor allem was man will ...



Ähnlich, wie du bei WoW als Priester heilen kannst und als Krieger eher nicht, legst du mit einem **Typ in C++** auch gleich fest, was du damit alles anstellen kannst. Schau dir einmal folgende Zeile an:

```
ehe = mann + frau;
```

Damit diese Zeile in C++ einen Sinn ergibt, müssen **mann** und **frau** vom **passenden Typ** sein, damit die Addition (+) und Zuweisung (=) verwendet werden kann und (eine) **ehe** dabei herauskommt.

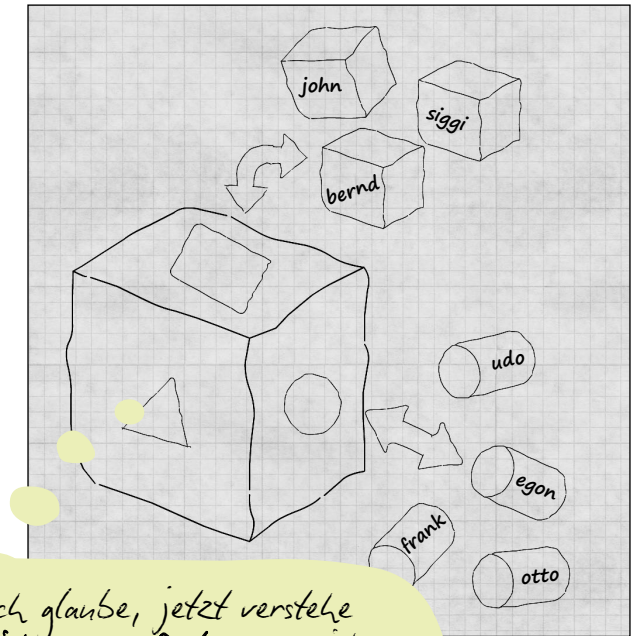
[Zettel]

Zwar kannst du die Kuh nicht in die Formenbox der fundamentalen Typen stecken, aber später lernst du noch, wie du mit den fundamentalen Typen eigene **benutzerdefinierte Typen** erstellen kannst. Du könntest quasi aus den grundlegenden Basistypen einen **eigenen Typ** Kuh erstellen.

## Mein Name ist Schrödinger

Um jetzt auf die Daten zugreifen zu können, benötigst du für jeden Typ einen Namen (genauer **einen Bezeichner**). Für einen solchen Namen kannst du beliebige Buchstaben, Ziffern und das Unterstrichzeichen (z. B. **\_huhu**) verwenden. Das **erste Zeichen** darf allerdings **keine Ziffer** (somit falsch: **8tung**) sein, und natürlich musst du auch hier auf die **Groß- und Kleinschreibung** achten. Somit sind **Nebelleben** und **nebelleben** zwei verschiedene Namen (egal, in welcher Richtung diese gelesen werden).

Damit jeder seinen Stein wiederbekommt, müssen eindeutige Namen verwendet werden.



[Schrödinger stolz]: Ich glaube, jetzt verstehe ich es. Zum **Speichern von Daten** muss ich einen für die Daten **passenden Typ** verwenden. Der Typ bestimmt dann, was ich mit den Daten machen kann (bspw. rechnen). Und damit ich auf diese Daten zugreifen kann, brauche ich einen **eindeutigen Namen!**

## Fundamentale und faule Typen

... wie aus grundlegenden Formen (Typen) eine Katze wurde.



Natürlich stellt dir C++ hier **grundlegenden Typen** zur Verfügung, in denen du deine Daten speichern kannst. Grundlegend heißt hierbei: **Ganzzahltypen**, **Gleitkommatypen** und Typen für **Zeichen**. Wenn du diese Typen besser kennengelernt hast, kannst du später daraus speziellere Typen basteln. Stell dir das einfach wie bei dem chinesischen Legespiel **Tangram** vor, bei dem du aus primitiven geometrischen Flächen unzählige Figuren legen kannst. Ebenso kannst du es in C++ mit primitiven Typen recht weit bringen und daraus besondere Typen zusammenlegen (bspw. auch eine Kuh).



Platon,  
mein Freund der Weisheit,  
du magst zwar die Wahrheit,  
das Schöne und das Gute lieben,  
aber ich will jetzt endlich  
programmieren und nicht  
philosophieren!?



Wie überall gibt es auch hier wieder Typen, die gar **nichts tun** wollen und sich jeder Information entziehen. Genauer handelt es sich hierbei um **void**. Der Typ kann sich gar nichts merken, und man kann ihn höchstens bei Funktionen gebrauchen. Der **sinnleere Typ** könnte zwar auch als Zeiger verwendet werden, aber das gilt als **böse** und sollte in C++ mit **Weihwasser** gebannt werden.

>> Gleich noch eine Weisheit, mein Guter

## Deklaration und Definition

Bevor du einen Namen in deinem C++-Programm verwenden kannst, musst du dem Compiler Bescheid geben, mit **welchem Typ** du daherkommst. Ein gegenseitiges Bekanntmachen gehört schließlich zum guten Ton, und ohne kommst du ohnehin nicht rein. Eine solche Verknüpfung zwischen Typ und Namen wird als **Deklaration** bezeichnet. Damit weiß der Compiler, welche Aktionen du auf das Speicherobjekt anwenden kannst (bspw. +, -, / usw.).

Eine einfache Deklaration kannst du dir so vorstellen:

```
Typ Name; *1
Typ Name01, Name02, Name03; *2
```

\*1 Gefolgt von **Typ**, gibst du den **Namen** für das Speicherobjekt an. Am Ende schließt du diese Deklaration mit einem **Semikolon** ab.

\*2 Getrennt durch ein einfaches **Komma**, lassen sich auch gleich mehrere Namen vom selben **Typ** deklarieren.

Meistens sind solche Deklarationen von Datentypen gleichzeitig auch **Definitionen**. Das heißt, es wird ein eindeutiges Objekt im Speicher erzeugt, dem alle nötigen Informationen (eben was der Typ alles kann) zugeordnet werden.

[Zettel]

Das Thema **Deklaration und Definition** ist nicht auf einfache Typen beschränkt. Auch konstruierte Typen und Funktionen müssen deklariert werden. Allerdings kommt es dabei häufig vor, dass du die Definition an einer anderen Stelle im Quellcode vornimmst als die Deklaration. Darüber brauchst du dir aber jetzt noch keine Gedanken zu machen.

## Ganzer Kerl dank ...

Offiziell bietet C++ derzeit verschiedene Größen von Ganzzahltypen (=Integer) an. Da wären (in aufsteigender Größe) **short int**, **int**, **long int** und **long long int**. Wie du an dem Namen **short int** schon herauslesen kannst, handelt es sich um den kleinen Bruder (Geschlecht der Typen ist leider nicht feststellbar) und bei **long int** um die große Schwester (im Sinne der Gleichberechtigung) von **int**. Die beiden, **short** und **long**, können auch ohne den Familiennamen **int** verwendet werden.

[Zettel]

Der Erfinder von C++, Bjarne Stroustrup will hierbei gerne eine vereinheitlichte Initialisierung sehen. Mit dem C++11-Standard wurde eine solche vereinheitlichte Initialisierung mit geschweiften Klammern eingeführt, welche sich neben einfachen Datentypen, die eben beschrieben werden, auch später für komplexere Typen wie Arrays, Strukturen, Klassen oder verschiedenen Behälter-Klassen (Container-Klassen) verwendet werden kann. Anstatt einer klassischen Zuweisung mit = wirst du hier in naher Zukunft auch häufiger die Version mit den geschweiften Klammern antreffen:

```
int dieter{36};
// Vereinheitlichte
// Initialisierung (C++11)
int helmut{dieter};
// Das geht auch mit C++11
```

\*4 **helmut** bekommt denselben Wert wie **dieter** zugewiesen.



>> Ein Blick in die Glaskugel

[Hintergrundinfo]

**long long int** ist ein extra **laaanger** Ganzzahltyp, der offiziell erst mit dem C++11-Standard dabei ist. Dies wollte ich nur erwähnen, falls du einen alten Compiler hast, der damit nichts anfangen kann.

Wenn du eine lokale Variable mit Namen anlegst, bedient sich dieser erst einmal selbst und greift einfach nach einem Wert, der da eben **zufällig rumliegt**. Dass ein solches Verhalten meistens **Probleme** mit sich bringt, sollte klar sein. Daher solltest du dich immer darum kümmern, dass eine Variable **gültige Werte** erhält. Hierbei gibt es mehrere Wege, von denen der grundlegendste wohl die **Zuweisung** mit dem = ist. Bei neueren C++11-Compilern kannst du auch die vereinheitlichte Initialisierung mit den geschweiften Klammern zwischen { } verwenden. Natürlich kannst du die Werte auch über den Stream **cin** und den Eingabeoperator in die Variable schieben:

\*1 **dieter** wird der Wert 36 zugewiesen.

\*2 **kleinDieter** ist **nicht initialisiert**, und **nenä** bekommt den Wert 99 zugewiesen.

```
int dieter = 36; *1
```

```
short kleinerDieter, nenä = 99; *2
```

```
long grosserDieter; *3
```

```
cin >> grosserDieter; *3
```

\*3 **grosserDieter** bekommt seinen Wert über den Eingabe-Stream (der Tastatur) **cin** mit dem Operator **>>** zugeschoben.

```
int helmut = dieter; *4
```

```
long schroedinger; *5
```

```
schroedinger = 1234; *5
```

\*5 Variablen müssen **nicht sofort** initialisiert werden.

# Zeichenfolgen von Ganzzahlen

Es gibt gewisse **Regeln**, die du bei den Zeichenfolgen einhalten musst, damit der Compiler die Erscheinung auch als Ganzzahl akzeptiert. Du wirst zwar anfangs meistens nur die dezimale Schreibweise (**Basis 10**) verwenden, aber daneben kannst du auch eine oktale (mit der **Basis 8**) und hexadezimale (mit der **Basis 16**) Schreibweise benutzen (das Zeichenliteral lassen wir mal außen vor). Die binäre Schreibweise mit dem 2er-System können Sie seit C++14 verwenden.

dezimal	0	1	100	255
oktal	00	01	0144	0377
hexadezimal	0x0	0x1	0x64	0xff
binär	0b0	0b1	0b01100100	0b11111111

Ein Beispiel mit oktalen, hexadezimalen und binären Gegenständen.



Verstehe, die oktale Schreibweise beginnt mit 0 und die hexadezimale Schreibweise mit 0x.

[Zettel]

Bei der **hexadezimalen Schreibweise** werden die Buchstaben a, b, c, d, e, f verwendet, um 10, 11, 12, 13, 14, 15 darzustellen.

# Positive und/oder negative Haltung und ein Endteil

Alle drei Ganzzahltypen haben eine **positive und negative** Haltung, was ihren Wertebereich betrifft. Wenn du einen Integer ohne weitere Angaben hinschreibst, kann dieser sowohl negative als auch positive Werte aufnehmen. Brauchst du allerdings einen ganzzahligen Wert, der nur **vorzeichenlos** sein soll, stellst du lediglich das Schlüsselwort **unsigned** voran, und schon hat der Typ seine negative Haltung fallen gelassen:

\*3 Hier kennzeichnest du mithilfe des Schlüsselwortes **signed** die Variable **neutralerTyp02** explizit als vorzeichenbehaftet. Auf das Schlüsselwort kannst du **verzichten**, weil ganzzahlige Typen ohne die Verwendung von **unsigned** immer vorzeichenbehaftet sind.

```
unsigned int positiverTyp; *1
int neutralerTyp01; *2
signed int neutralerTyp02; *3
```

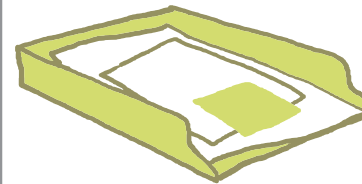
\*1 Die Variable dieses Typs kann wegen des Schlüsselwortes **unsigned** **keine** negativen Werte speichern.

\*2 Die Variable dieses Typs speichert **negative und** positive Werte.

# Von Us und Ls ...

[Ablage]

Hier noch eine Schnellnotiz, wie du deine Zeichenfolge von Ganzzahlen noch komplizierter machen kannst. Hängst du am Ende ein **U** an die Zahl, gilt diese Zahl explizit als **unsigned** (z. B. **1234U**). Hängst du ans Ende ein **L**, wird die Zahl explizit als **long int** betrachtet (z. B. **1234L**). Kombiniert du beide miteinander, kannst du auch ein **unsigned long int** daraus machen (z. B. **1234UL**). Und wenn du die ganz laaangen Typen wie **long long (int)** oder **unsigned long long (int)** brauchst, hängst du einfach ein **LL** (für **long long**; z. B. **1234LL**) oder ein **ULL** (für **unsigned long long**; z. B. **1234ULL**) hinten dran. Hängst du nichts hinten an, dann handelt es sich um ein **int**. In der Praxis findest du die Verwendung der **Us** und **Ls** eher selten, weil der Compiler oft schon selbst erkennt um was für einen Ganzzahltyp es sich bei einem Literal handelt.



# Die Sache mit der Wahrheit ...

Wenn du herausfinden willst, ob deine Freundin dir treu ist, kannst du den **Booleschen Typ** verwenden. Der Typ **bool** ist nur darauf spezialisiert, ob etwas **wahr (=true)** oder **unwahr (=false)** ist. Damit kannst du die verschiedensten Werte bzw. Ausdrücke auf Wahrheit hin testen.

Ein einfaches Beispiel:



\*1 Wenn **dieter** und **eva** den gleichen Wert hätten, wäre der Wert von **treu** gleich **true**. Im vorliegenden Fall haben diese beiden Variable unterschiedliche Werte. Daher hat **treu** hier den Wert **false**. Sieht wohl nicht gut aus für die beiden ...

\*2 Hier ist **ichLiebeDich** auf jeden Fall **true**, weil eine ganze Zahl ungleich 0 **immer** nach **true** konvertiert wird. Hierbei wird der Wert 1234 nach 1 (**=true**) konvertiert.

\*3 **liebe** bekommt hier den Wert 1 zugewiesen, weil ein Boolescher Wert eben **nur** die Werte **true** (1) oder **false** (0) enthalten kann.

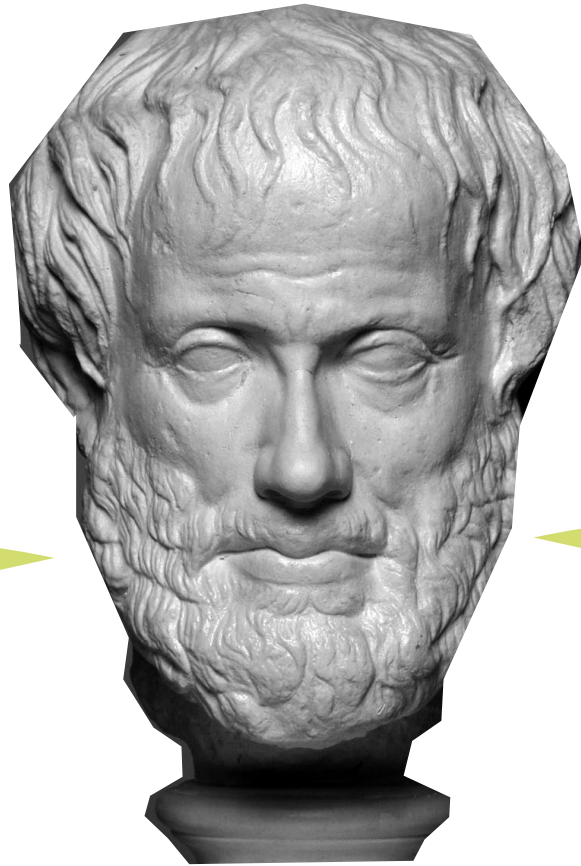
\*4 **iLoveYouBaby** hingegen ist **false**, weil ein Wert von 0, was **wahreLiebe** eben ist, **immer** nach **false** konvertiert wird.

```
int dieter=1974, eva=1980; *1
bool treu = dieter==eva; *1
bool ichLiebeDich = 1234; *2
int liebe = ichLiebeDich; *3
int wahreLiebe = 0;
bool iLoveYouBaby = wahreLiebe; *4
```

Nicht darum nämlich, weil unsere MEINUNG, du siehst weiß, wahr ist, bist du weiß, sondern darum, weil du weiß BIST, sagen wir die Wahrheit, indem wir dies behaupten. Also sag mir doch, du weißer Mensch, WAS soll ich mit **bool** anfangen?



Werter Schrödinger, zu argwöhnen, das Seiende sei anzweifelbar, ist der erste Schritt, zwischen wahr und unwahr zu unterscheiden.



Um deine Anfrage als Wahres zu beantworten, Folgendes: Du wirst die Logik erst erkennen, wenn du weiter an deinem Sein arbeitest und später in Logikanweisungen oder Funktionen zwischen Wahrem und Unwahrem unterscheiden willst.

## Was nehmen wir für einen Typen?

Nachdem du die **Typen für Ganzzahlen** kennst, wird es Zeit, dass du diese auch mal benutzt. Du solltest dir dabei jetzt noch keine Gedanken bezüglich der **Größe** der Typen machen. Das klären wird später noch.



[Einfache Aufgabe]

Erstelle ein einfaches Programm, welches deinen Kontostand abfragt und auch, wie viel Gramm Wurst du vom Metzger Meier gerne hättest. Überlege dir, was du bei den beiden Variablen **kontostand** und **gramm** beachten solltest.



### Hierzu eine mögliche Lösung

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

```
int main()
{
    int kontostand; *1
    unsigned int gramm; *2

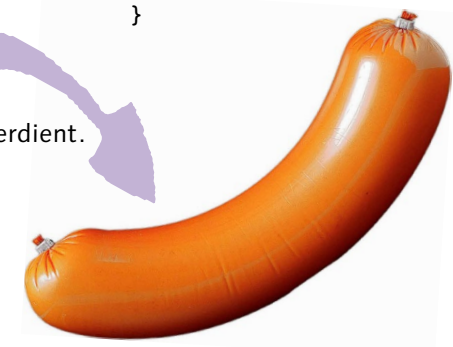
    cout << "Dein Kontostand : ";
    cin >> kontostand; *3
    cout << "Wieviel Wurst darf es ein: ";
    cin >> gramm; *3
    cout << "Kontostand      : " << kontostand
        << "\nWurstaufschnitt : " << gramm
        << " gramm" << endl;
    return 0;
}
```

\*3 Über den Eingabe-Stream **cin** schiebst du mithilfe der >>-Operatoren die Werte von der Tastatur in die entsprechenden Variable.

\*1 Hier könntest du genauso gut den Typ **short** oder **long** verwenden (hängt natürlich von der persönlichen Finanzlage ab). **Wichtig** ist auf jeden Fall, dass hier ein Typ **mit Vorzeichen** verwendet wird, weil man das Konto ja auch überziehen soll, damit die Bank auch was dran verdient.

\*2 Hier brauchen wir mit **unsigned** einen **vorzeichenlosen Typ**, weil es eben keine „-100 g“ Wurst gibt.

[Belohnung] Jetzt hast du dir eine Belohnung verdient.



## Die Welt der ganzen Kerle

Neben den ganzzahligen Kerlen hast du jetzt auch erfahren, wie du eigene Variablen anlegen und verwenden kannst. Das gilt übrigens auch für die **kommenden Typen**, die dir noch begegnen werden. Hier noch eine kleine **Checkliste**, was du dir für die Verwendung von Typen hinter die Ohren schreiben solltest:

- ☑ Du brauchst einen **eindeutigen, gültigen Namen** (Bezeichner), der aus beliebigen Buchstaben, Ziffern und dem Unterstrichzeichen bestehen darf. Dabei wird auch zwischen **Groß- und Kleinschreibung** unterschieden!
- ☑ Wie bei Legobausteinen gibt es in C++ zunächst nur **fundamentale Typen** (Basistypen). Aus diesen Typen kannst du später **komplexere** erstellen (ja, auch Klassen).
- ☑ Jeder fundamentale Typ (neben den Integern gibt es noch andere) hat ein seiendes Ding von etwas (genauer **Entität**), eine Liste von **Fähigkeiten** eben, was mit dem Typ möglich ist und was nicht.
- ☑ Bevor du diesen Namen verwendest, musst du dem Compiler Bescheid geben, mit welchem Typ du gehen willst (**Deklaration**). Auch zu den ganzen Typen, die dir in diesem Kapitel vorgestellt wurden, kannst du noch Folgendes in dein Gehirn einfügen (einfaches Copy & Paste eben):
- ☑ Ganzzahltypen gibt es mit **short (int)**, **int** und **long (int)** und **long long (int)** in **drei verschiedene Größen**.
- ☑ Wohnt der Typ im selben Haus (lokaler Typ), hat er ohne eine direkte Initialisierung zunächst **keinen gültigen Wert**.
- ☑ Standardmäßig haben alle Ganzzahltypen ein **Vorzeichen (signed)**. Soll das Vorzeichen verschwinden, musst du das Schlüsselwort **unsigned** davorsetzen.

Dann war da noch der Aristoteles-Typ mit der Wahrheit. **bool** hieß der und kann nur die Werte 1 und 0 darstellen. Also alles, was ungleich 0 ist, betrachtet **bool** als 1 oder eben als Synonym **true**, und alles 0-wertige ist eben 0 oder das Synonym **false**. Ich weiß zwar immer noch nicht genau, wozu dieser Typ eigentlich gut ist, aber das wird schon noch kommen. Kann ich jetzt endlich schlafen gehen

...?!



[Belohnung/Lösung]

Jawoll!



## Was für den einen das Komma, ist für den anderen der Punkt ...

*Komma?  
Wo ist hier das  
Komma?*

In C++ wird die **US-SCHREIBWEISE** für die Gleitkommatypen verwendet. Und da wird das Pünktchen dem Komma vorgezogen. Folglich könnte man zwar auch von **GLEITPUNKTTYPEN** reden, was allerdings dann wieder nicht unserer Kultur passt, weil es das bei uns halt nicht gibt. Daher wurde **FLOATING POINT** einfach an die lokalen Gegebenheiten angepasst und als **GLEITKOMMATYP** übersetzt. Denk also auch daran, dass du bei der Eingabe einen **PUNKT** statt eines Kommas verwendest. Und, ja, theoretisch ist es möglich, den Punkt zum Komma umzuoperieren, aber in der Praxis ist das eher unüblich.

Auch bei Zahlen mit **Kommas** wirst du mit verschiedenen Größen versorgt. Mit **float** für einfache Genauigkeit, **double** für doppelte Genauigkeit und **long double** für erweiterte Genauigkeit stehen dir drei fundamentale Typen zur Verfügung. Die exakte **Genauigkeit** ist übrigens abhängig davon, wie die Compilerbauer diese implementiert haben.

**Lieblingstyp** und vom Compiler bevorzugter Typ ist immer **double**. Auch bei den Kommatypen gibt es einige **Regeln**, damit die Zeichen am Ende auch als Kommatyp angenommen werden. Hier einige Beispiele:

123.456 .33 0.33 2. 2.0 2.2e10  
2.22e-14

Steht vor oder nach dem Pünktchen eine **0**, kannst du diese auch **weglassen** (bspw. macht der Compiler aus **.5** gleich **0.5** oder aus **3.** ein **3.0**). Ein Leerzeichen darf allerdings bei der Zeichenfolge eines Gleitkommatyps **nicht** stehen.

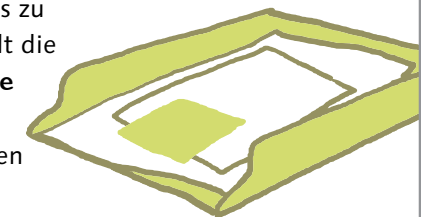
## Von fs und Ls ...

[Ablage]

Wie auch bei den Ganzzahlen kannst du auch hier ans Ende der Gleitkomma-Zeichenfolge noch einen **Buchstaben** setzen, um aus dem Liebling **double** etwas anderes zu machen. Hängst du den Buchstaben **f** ans Ende (bspw. **3.1415f**, **3.7e-3f**) stellt die Zeichenfolge einen **float** dar. Mit einem **L** am Ende kannst du ein **long double** anzeigen (bspw. **3.145L**, **3.7e-3L**).

Das Thema **Initialisieren und Bekanntmachen** wurde bereits bei den ganzen Typen behandelt und gilt hier natürlich **genauso**. Einige Beispiele:

```
float mynameislittledot = .535; // 0.535
double daddysdarling = 3.1415;
long double extendenddot = 1.9e-3f;
double Cplusplusstandard{123.456}; // mit C++11 möglich
```



# Das Pünktchen in der Werkstatt



Zu dem Thema willst du sicherlich auch ein Beispiel schreiben, wie du einen Gleitkommatyp von der Tastatur einlesen kannst:

**Gib ihm eine Zahl  
mit Pünktchen ...**

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

```
int main()
{
    double willPuenktchen; *1
    cout << "Gib ihm eine Zahl mit dem Pünktchen: ";
    cin >> willPuenktchen; *2
    cout << willPuenktchen << endl;
    return 0;
}
```

\*2 Es wird alles eingelesen, solange die **Regeln** der gültigen Zeichenfolge (Literal) eines Gleitkommatyps eingehalten werden. Gibst du bspw. 3.55 ein, bricht **cin** ab dem Komma ab, und **willPuenktchen** enthält 3.0. Nicht vergessen: **Pünktchen statt Kommas!**

\*1 Bei diesem sinnfreien Beispiel kannst du genauso gut den Typ **float** oder **long double** statt **double** verwenden. Ich habe mich halt gleich für **Daddys Liebling** entschieden.

Schön und gut, aber ich weiß jetzt immer noch nicht, was ich im Fall der Fälle für einen Gleitkommatyp auswählen soll!?



Tut mir leid, aber hier kann ich dir vorerst keine feste Empfehlung geben. Das Problem ist leider, dass es von der **Implementierung** abhängt, wie die Kommatypen dran und drin sind. Daher ist der Sinn von einfacher, doppelter und erweiterter Genauigkeit auch abhängig von der Einpflanzung der Typen im Compiler. Hier bleibt dir zunächst nichts anderes übrig, als dich mehr mit dem theoretischen Thema Gleitkommazahlen auseinanderzusetzen (**Wikipedia** ist dein Freund). Wenn du dazu keine Lust hast, kannst du ja immer noch Daddys Liebling **double** verwenden und schauen, was passiert.

# Am Ende war das Pünktchen ...

In C++ wird nicht das Komma, sondern das Punktzeichen verwendet, und es gibt die drei Typen **float**, **double** und **long double** mit einer unterschiedlichen Genauigkeit.

Ach, Schrödinger, schonmal was von Gleitkommazahlen gehört? **double** ist ja gut und schön, aber über Gleitkommazahlen musst du dich schlaue machen. Ich erklär's dir gerne ...

Ach nee, ist kein Problem für mich.

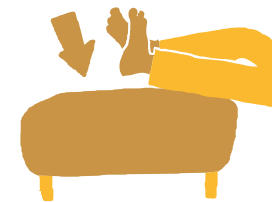
An die Arbeit geht's. Theorie kann warten.

[Einfache Aufgabe]

Findest du die Fehler in dem folgenden Codeausschnitt?

```
float floaty_1 = 3. 3334f;
float floaty_2 = ,3335f;
double doubly = 3.3e10 + 10;
long double ldoubly = 2.34e - 11;
```

Bin gespannt, ob du es lösen konntest. Hier die Lösung



[Belohnung/Lösung]

Jetzt ist aber eine dicke Belohnung drin: die neue Blu-ray „Angriff der achtbeinigen Monster“!

\*1 Hier steht ein **Leerzeichen** nach dem Punkt, was falsch ist.

\*2 In dieser Zeile wird ein **Komma** statt eines **Punktes** verwendet.

```
float floaty_1 = 3.3334f; *1
float floaty_2 = .3335f; *2
double doubly = 3.3e10 + 10; *3
long double ldoubly = 2.34e-11; *4
```

\*3 Hier ist rein syntaktisch **kein Fehler** vorhanden. Die Addition mit 10 ist kein Fehler. Aber ob das hier so gewollt war, bleibt unklar?!

\*4 Hier ist es aber ein Fehler, weil der **Exponent** (e) noch Ziffern benötigt, damit es sich um eine gültige Zeichenfolge einer Gleitkommazahl handelt. Beide **Leerzeichen** vor und nach dem Minussymbol sind in dem Fall also falsch.

## Zeichensalat

Wenn du einzelne **Zeichen** darstellen willst, musst du den **Binärcode** (Darstellungen 0 und 1) verwenden. Nein, so weit musst du glücklicherweise nicht gehen. Für Zeichentypen stellt dir C++ den fundamentalen Typ **char** zur Verfügung. Meistens (muss aber nicht sein!) ist **char** mit 8 Bits ausgestattet. In diesen 8 Bits lassen sich somit theoretisch 256 Zeichen ( $2^8 = 256$  verschiedene Bit-Darstellungen von 0 und 1) speichern.

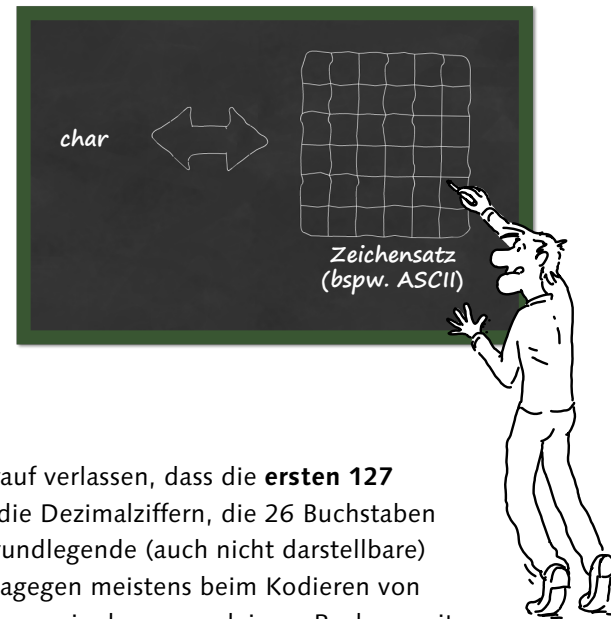
Um hier nicht wie in einer Art Zweistromland mit einer Kolonne von 1 und 0 einen Buchstaben darstellen zu müssen, existiert ein **Zeichensatz** dafür. Du kannst dir diesen wie eine Tabelle vorstellen, in der **char** sein Zeichen für einen bestimmten Code erhält und benutzen kann. Meistens (muss auch nicht sein!) wird hier der **ASCII-Code** verwendet.

[Zettel]

Es gibt ausgefallene Rechner, wo ein **char** nicht auf acht Beinen daher kommt, aber ich hoffe, dass dir solche „Monster“ nicht über den Weg laufen.



Der Zeichentyp **char** kann ein „Zeichen“ aus einem Zeichensatz speichern



[Hintergrundinfo]

In der Praxis kannst du dich meistens darauf verlassen, dass die **ersten 127 Zeichen** (0 bis 127) im (US-)ASCII-Code die Dezimalziffern, die 26 Buchstaben des englischen Alphabetes und andere grundlegende (auch nicht darstellbare) Zeichen enthalten. Schwieriger sieht es dagegen meistens beim Kodieren von landestypischen (**diakritischen**) Zeichen aus, wie du es von deinem Rechner mit den Umlauten her kennst. Hierbei musst du dich dann leider mit dem verwendeten Zeichensatz auf deinem System auseinandersetzen. Als universelle Lösung könntest du UTF-8-Literale wie `u8'\u00fc'` oder die Locale-Komponente von C++ verwenden.

## Doch ein ganzer Kerl?

Damit der Compiler ein Zeichen auch als solches annimmt, musst du es zwischen **einzelne Anführungszeichen** setzen (z. B. `'A'` oder `'Z'`). Genau genommen ist allerdings eine solche Zeichenfolge (genauer **Zeichenliterale**) allerdings nur ein symbolischer fester Wert für den ganzzahligen Wert des Zeichens im Zeichensatz. Verwendet dein Rechner (höchstwahrscheinlich) den ASCII-Zeichensatz, so ist der Wert `'A'` gleich dem Wert 65. Somit hätte folgende Verwendung von **char** dieselbe Bedeutung:

```
char Symbol_fuer_A = 'A';  
char Dezimal_fuer_A = 65;
```

Dann ist es also egal, ob ich das Ding mit den Hochkommata oder direkt den dezimalen Wert verwende?

**Nur**, wenn du dir sicher sein kannst, dass auf jedem Rechner derselbe Zeichensatz läuft, schon, was aber laut Murphys Gesetz nicht immer der Fall ist. Wenn auf einem System ein anderer Zeichensatz verwendet würde, wäre die Hochkomma-Variante **portabler**. Denn dann müsstest du dich nicht um den tatsächlichen Wert des Zeichens kümmern, da dir das ja in dem Fall der Zeichensatz abnähme.

## Positiver oder negativer Typ

Hurra, die Werte zwischen den Bereichen **0 bis 127** schaffen in der Regel kaum Probleme. **Alle anderen** Werte außerhalb des Bereiches leider schon. Sätze wie „**Schrödinger könnte grüne Grütze rühren**“ könnten somit auf unterschiedlichen Rechnern zu interessantem Zeichensalat führen. Mit UTF-8 und Konvertierungen kann man das Problem in C++11 ganz Allgemein und portabel schaffen.



Es kommt noch schlimmer mit den Zeichen

...

[Achtung]

Bei **char** hängt es von den Compilerbauern ab, ob **char** als **signed** (–128 bis 127) oder als **unsigned** (0 bis 255) implementiert ist. Das ist natürlich ein weiterer Grund, bei **char** auf **Dezimalzahlen** zu verzichten.

## Turmbau zu Babel

Nachdem es C++ also **nicht kümmert**, welcher Zeichensatz verwendet wird, musst du dich selbst mit dem Thema herumschlagen. Aus purem Egoismus (oder Patriotismus) der US-amerikanischen Ingenieure wurde der ASCII-Zeichensatz auf den ersten **sieben Bits** verteilt und das achte Bit als **Paritäts-Bit** verwendet. Wir Europäer hatten somit keinen Platz mehr für die landestypischen Zeichen wie **üäößÜÖÄ** oder die anderen europäischen landestypischen Zeichen.

Eine Gruppe von Leuten (die ISO) haben sich darum gekümmert. Sie haben den ASCII-Zeichensatz auf **8 Bits** aufgeböhrt und diesen Zeichensatz unter Bezeichnungen wie ISO-8859-1, ISO-8859-2 usw. zusammengefasst. Unsere Umlaute findest du daher in **ISO-Latin-1** wieder.

Allerdings ist es nicht damit getan, den Zeichensatz der Eingabeaufforderung einfach auf ISO-Latin-1 umzustellen, weil es sonst wieder Probleme mit der viel weiter verbreiteten **UTF-8-Kodierung für Unicode** gibt, die auf modernen Computern zur Darstellung von Umlauten verwendet wird. Hinzu kommt noch die Eingabeaufforderung von Windows, welche aus Kompatibilitätsgründen immer noch am **alten IBM-PC-Zeichensatz** festhält, in dem der Dezimalwert der Zeichen wieder einen anderen Wert hat.



## Wo wir schon beim Thema sind ...



[Hintergrundinfo]

Die **Unicode-Zeichen** haben natürlich nicht mehr Platz in einem Byte (was auf 256 Zeichen beschränkt ist), also in **char**. Daher wird für solche Zeichen statt auf **char** auf den breiteren Typen **wchar\_t** zurückgegriffen. Die Größe von **wchar\_t** wiederum hängt davon ab, wie die Compilerbauer diese implementiert haben (meistens mit 2 oder 4 Bytes). Bei der Verwendung von solchen breiten Zeichen musst du ein **L** vor das Zeichen stellen (bspw. **L'A'**).

## Zum Flüchten ...

Neben den für dich **sichtbaren Zeichen** gibt es noch Zeichen, die zwar nicht dargestellt werden, die aber trotzdem etwas bewirken. Solche **Steuerzeichen** werden mit einem umgekehrten Schrägstrich (auch Backslash genannt), gefolgt von einem Zeichen mit fester Bedeutung geschrieben. Bekannter Vertreter ist bspw. das Auslösen einer neuen Zeile mit dem Zeichen **'\n'**.

## Unicode-Unterstützung

Ein paar mal habe ich dir ja bereits den Brocken mit den UTF-8-Zeichen hingeworfen. Und da ich eben **wchar\_t** beschrieben habe, was ja auch nicht so recht was Gescheites ist, weil eben die Länge nicht exakt spezifiziert ist und es somit wieder Probleme bei der Portierung auf anderen Rechnern geben kann, möchte ich dir kurz die drei neuen Unicode-Kodierungen UTF-8, UTF-16 und UTF-32 vorstellen, welche in C++11 eingeführt wurden. Dafür wurden sogar eigens für UTF-16 der Typ **char16\_t** und für UTF-32 der Typ **char32\_t** zwei neue Typen hinzugefügt.

Kodierung	Typ	String-Literal
UTF-8	char	u8"Ein UTF-8-String"
UTF-16	char16_t	u"Eine UTF-16-String"
UTF-32	char32_t	U"Ein UTF-32-String"

Zeichenfolge	Bedeutung
\a	akustisches Signal
\b	Rückschritt/Backspace
\f	Seitenvorschub
\n	Zeilenende
\t	Tabulator (horizontal)
\v	Tabulator (vertikal)
\ddd	oktale Zahl (bspw. \033 = ESC)
\xhh	hexadezimale Zahl
\uXXXX	für Zeichen eines größeren Zeichensatzes (bspw. Unicode)
\UXXXXXXXX	

Verwenden kannst du solche Unicode-Zeichen (Unicode-Codepunkt) mit der Zeichenkombination **\u** bzw. **\U**, welche du in einer String-Literale einbettest. Hinter **\u** muss eine **16-Bit Hexadezimalzahl** stehen, wohin gegen hinter einem **\U** eine **32-Bit Hexadezimalzahl** erwartet wird. Hier drei solche Beispiele, welche alle **\u00f6** verwenden, welches dem Umlaut ö entspricht:

```
cout << u8"Schr\u00f6dinger\n"; // UTF-8
cout << u"Schr\u00f6dinger\n"; // UTF-16
cout << U"Schr\u00f6dinger\n"; // UTF-32
```

[Notiz]

Eine praktische UTF-8-Tabelle findest du hier: <http://www.utf8-zeichentabelle.de/>  
Willst du hingegen direkt nach einem bestimmten Unicodezeichen suchen, hilft dir diese Webseite weiter: <http://www.isthisthingon.org/unicode/index.php>

# Zeichen für die Welt

So, nachdem du die Grundlagen zu den Zeichen kennengelernt hast, ist es nun an der Zeit, dass du hiervon welche auf die Welt loslässt. Daher darfst du jetzt endlich auch wieder deinen teuer erworbenen **Rechner anwerfen** und ein paar Zeilen mit Code eintippen:

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

```
int main()
{
    char einA = 'A', einZ = 'Z'; *1
    cout << einA << '-' << einZ << '\n';
    cout << int(einA) << '\t' << int(einZ) << '\n'; *2

    char A65 = 65, Z90 = 90; *3
    cout << A65 << '\t' << Z90 << '\n';
    return 0;
}
```

\*1 Hier wird jeweils einmal das Zeichen 'A' in **einA** und das Zeichen 'Z' in **einZ** gespeichert und dann mithilfe des <<-Operators nach **cout geschoben** und auf dem Bildschirm ausgegeben.

\*2 Mithilfe des Ausdrucks **int(ch)** kannst du den **ganzzahligen Wert** der Zeichen 'A' und 'Z' auf dem Bildschirm ausgeben.

\*3 Anders herum funktioniert dies genauso. Hier wird an den **char**-Variablen **A65** der ganzzahlige Wert 65 und an **Z90** der ganzzahlige Wert 90 übergeben. Der Wert **65** ist auf der **ASCII-Tabelle** der Buchstabe **A** und der Wert **90** ist ein **Z**. Allerdings setzt diese Art, ein Zeichen als ganzzahligen Wert zu verwenden, voraus, dass auf dem Rechner der ASCII-Zeichensatz läuft. Ist dies nicht der Fall, wird irgendein Zeichen ausgegeben, welches eben diesen dezimalen Wert im Zeichensatz hat. Beim **EBCDIC-Zeichensatz** wäre bspw. 65 gar nicht belegt und 90 das Ausrufezeichen (!).

Das ist eigentlich mit **Unicode** schon passiert bzw. im Gange. Unicode ist ein Zeichensatz, der versucht, **weltweit alle** bekannten **Zeichen** in einem Zeichensatz **zusammenzufassen**. Das ist gar nicht so einfach, allein wenn du an die unzähligen Zeichen in der chinesischen oder koreanischen Schrift denkst. Und da eben die enorme Menge an Zeichen keinen Platz in einem **char** haben, welches ja auf 256 Zeichen beschränkt ist, kannst du hierfür **char16\_t** oder **char32\_t** verwenden, mit denen du 65.536 und mehr als 4 Milliarden Zeichencodes unterscheiden kannst. Früher hat man dazu **wchar\_t** verwendet, was aber eben mal 2 oder 4 Bytes groß ist.



[Notiz]

Denk daran, ein **char** oder **wchar\_t** selbst speichert keine Zeichen, sondern letztendlich auch wieder nur Ganzzahlen, die ihre Bedeutung erst mit dem auf dem Rechner befindlichen **Zeichensatz** erhalten. Einzelne Zeichen wie 'A' sind letztendlich nur **symbolische Konstanten** für den ganzzahligen Wert des Zeichens aus dem Zeichensatz des Rechners.



[Code bearbeiten]

**wchar\_t** ist kein einfacher 1-Byte-Typ mehr, weshalb der Stream **cout** diese Zeichen nicht mehr verarbeiten kann. Wenn du **breite Zeichen** verwenden willst, musst du diese durch die **w**-Streams schieben. In deinem Fall wäre das also **wcout** statt **cout**.

*Hrumpf!*  
Ich habe das Beispiel eben mit **wchar\_t** getestet. Jetzt werden nur noch Ganzzahlen statt der Zeichen ausgegeben!?

*Das mit den Zeichensätzen regt mich total auf. Warum kann man nicht alles in einen Zeichensatz stellen, damit ich mich nicht damit herumschlagen muss?*

Nun ja, Schrödinger! Das Thema mit den Zeichensätzen mag dir am Anfang wie der Turmbau zu Babel vorkommen. Aber mittlerweile sieht es gar nicht mehr so schlimm aus. Spätestens seit dem C++11-Standard ist es mit den neuen Typen **char16\_t** und **char32\_t** und der Unicode-Welt erheblich einfacher geworden, weil hier mit UTF-8, UTF-16 und UTF-32 gleich drei Unicode-Kodierungen unterstützt werden.





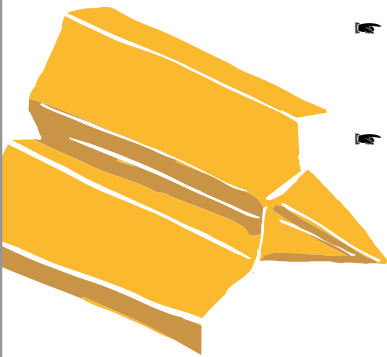
## Erste „Zeichen“ für eine Pause

Schrödinger ist frustriert, er hätte nicht gedacht, dass er sich um die Verwendung von einfachen Zeichen überhaupt einen Kopf machen muss. Im Augenblick hat er das Gefühl, dass er die Kontrolle über die Zeichen verliert.



Na gut, Schrödinger! Fassen wir mal nur die wichtigen Fakten zu **char** zusammen, damit du das Ganze wieder klarer siehst:

- Zum Speichern **einzelner Zeichen** kannst du **char** verwenden.
- **char** selbst speichert **keine Zeichen** im eigentlichen Sinne, sondern sucht das passende Zeichen aus einem Zeichensatz auf dem Rechner anhand eines dezimalen Wertes aus.
- Einzelne Zeichen können zwischen **einzelnen Anführungszeichen** oder als **Ganzzahl** mit dem Wert des Zeichens aus dem Zeichensatz verwendet werden.
- Die Verwendung von einzelnen Zeichen wie **'T'** ist im Grunde nur eine **symbolische Konstante** für den ganzzahligen Wert im Zeichensatz des Rechners.
- C++ schreibt nicht vor, welcher **Zeichensatz** verwendet werden soll. Zwar kannst du fast sicher sein, dass der **ASCII-Zeichensatz** auf deinem Rechner unterstützt wird, aber bei der Verwendung von Zeichen darüber hinaus (bspw. Umlaute) wird es dir oft passieren, dass auf dem einen Rechner alles glattgeht, während auf einem anderen Rechner **Zeichensalat** ausgegeben wird.
- Solltest du breitere Zeichen als **char** benötigen, findest du mit **wchar\_t** einen Typ dafür. Anlog musst du dann auch die Streams für breite Zeichen verwenden (**wcout, wcerr, wcin**).
- C++11 unterstützt mit UTF-8, UTF-16 und UTF-32 drei Unicode-Kodierungen. Für UTF-16 wurde der Typ **char16\_t** und für UTF-32 der Typ **char32\_t** eingeführt. Die neue Unicode-Kodierung ist natürlich **wchar\_t** vorzuziehen.



[Einfache Aufgabe]

Bevor du deine **wohlverdiente Pause** mit deinen WoW-Freunden verbringen kannst, **will ich** noch, dass du die Fehler im folgenden Codeausschnitt findest.

[Zettel]

```
char A = 'A';
char B = A;
char C = D;
char E = 66;
char F = 333;
char G = '/n';
```

```
cout << A << ' ' << B << ' '
     << C << ' ' << E << ' '
     << F << ' ' << G;
```

**\*1 Kein Fehler!**  
Der Variablen **B** wird der Inhalt von Variable **A** (also ein **'A'**) zugewiesen.

**\*2 Fehler!** Hier wurde versucht, an **C** eine Variable mit dem Bezeichner **D** zu überweisen, welcher nicht existiert. Da es keinen solchen Namen gibt, war hier wohl das Zeichen **'D'** gemeint.

```
char A = 'A';
char B = A; *1
char C = D; *2
char E = 66; *3
char F = 333; *4
char G = '/n'; *5
```

**\*3 Kein Fehler!** Nach dem ASCII-Zeichensatz entspricht der ganzzahlige Wert 66 dem Zeichen **'B'**.

**\*4 Teilweise ein Fehler!**  
Hier wird ein Überlauf produziert und die Ganzzahl verwendet, welche der entsprechenden Bit-Darstellung entspricht. Der Compiler sollte dich allerdings davor warnen.

**\*5 Hier ist der Schrägstrich verkehrt herum.** Statt eines Backslashes (**\**) wird hier ein normaler Slash (**/**) verwendet.



[Belohnung/Lösung]

So, jetzt kannst du dich ins Vergnügen stürzen.

## Auf die Größe kommt es an ...

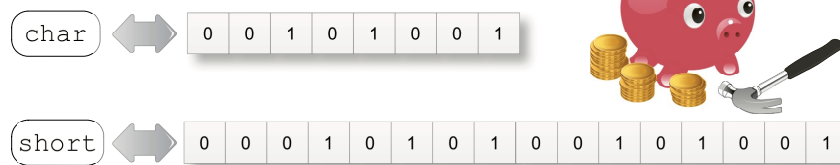
Wenn du stundenlang WoW spielst oder wenn du dich in der realen Welt zurechtfinden willst, braucht es **gewisse Grenzen**. In deinem konkreten Fall kommst du ohne räumliche oder zeitliche Grenzen nicht aus. Stell dir mal ein Leben ohne diese beiden Grenzen vor! Unmöglich! So ist es auch mit unseren Datentypen, die auch nicht überirdisch und damit an bestimmte Grenzen und **Größen** gebunden sind.

Genau genommen hängen diese Grenzen von der **Aneinanderreihung einzelner Bits** ab, welche die Zustände 0 und 1 darstellen können. Dadurch lassen sich verschiedene Werte darstellen.

Oh je,  
jetzt fängt er  
auch noch  
mit Haarspalterei  
an ...!

**Keine Sorge**, ich fasse mich kurz. Nimm als Beispiel den Typ **char**. Du hast bereits erfahren, dass **char** 256 verschiedene Werte darstellen kann (egal, ob es jetzt mit oder ohne Vorzeichen implementiert wurde). Diese Werte ergeben sich aus den **acht aneinandergereihten einzelnen Bits** ( $2^8 = 256$ ). Das ist in etwa wie bei einem Zahlenschloss mit acht Stellen, bei dem **jede (Bit-)Stelle** den Wert 0 oder 1 haben kann. Hierbei gibt es 256 verschiedene Möglichkeiten, das Schloss zu knacken. Wie dieser Wert letztendlich interpretiert wird (Zeichen, Ganzzahl, Gleitkommazahl oder Wahrheitswert), gibst du erst mit dem Datentyp an.

Je mehr Bits aneinandergereiht sind, desto größer sind die darstellbaren Werte, oder je größer das Sparschwein, desto mehr Geld passt rein.



[Achtung]

Beachte bitte, dass **1 Byte nicht** zwangsläufig aus **8 Bits** bestehen muss, auch wenn alle das vielleicht immer behaupten und es bei deinem und den meisten anderen Rechnern der Fall sein dürfte. Es gibt z. B. Rechner, bei denen ein **char** mit 32 Bits implementiert ist.

## Je größer, desto besser

Sicherlich willst du jetzt endlich **die Grenzen** der einzelnen fundamentalen Typen wissen? Ehrlich gesagt, ich kann sie dir nicht genau nennen, weil Dinge wie die Größe abhängig davon sind, wie diese implementiert sind. Anstatt dir hier also eine Liste mit Typen aufzuzählen, wie sie vielleicht sein könnten, bekommst du lieber **Steckbriefe** zu den Typen:

### Wahrheitswert: **bool**

Da **bool** nur **zwei Zustände** speichern kann (**true/1** und **false/0**), sollte eigentlich **1 Bit** ausreichen, um einen Wert zu speichern. Allerdings ist die kleinste adressierbare Einheit eben **1 Byte**, weshalb **bool** meistens mit dieser Größe implementiert ist. Es ist aber durchaus möglich, dass **bool** aufgrund besserer Zugriffsgeschwindigkeit dieselbe Größe wie die Prozessorarchitektur (32 oder 64 Bits) besitzt.

### Zeichen: **char**, **wchar\_t**, **char16\_t** und **char32\_t**

Die Größe von C++-Objekten wird immer als Vielfaches von der Größe eines **char** angegeben. **char** ist in der Regel **immer 1 Byte** groß und kann somit 256 verschiedene Zeichen darstellen. Genügend für den ASCII-Code und auch deutsche Umlaute. Der Typ **wchar\_t** für breitere Typen (bspw. **Unicode**) ist gewöhnlich mit 2 oder 4 Bytes Größe implementiert. Denk daran, dass es sich bei den beiden Typen trotzdem um Ganzzahlen handelt. Besser als **wchar\_t** für Unicode-Zeichen sind natürlich die neuen in C++11 eingeführten Typen **char16\_t** und **char32\_t** welche zur Zeichendarstellung von UTF-16 und UTF-32 verwendet werden, weil diese beiden Typen wesentlich portabler sind. So garantiert **char16\_t** eine Breite von mindestens 16 Bit wie auch **char32\_t** mindestens eine Breite von 32 Bit garantiert. Bei **wchar\_t** ist dies nicht gegeben!

### Ganzzahlen

Der **natürlichste Typ** für Ganzzahlen ist **int**, weil dieser gewöhnlich zur Größe der **Ausführungsumgebung** passt. Bei den handelsüblichen 32-Bit-Rechnern sind dies somit 4 Bytes. Bei 16-Bit-Systemen sind es nur 2 Bytes. Bei den anderen Ganzzahltypen gilt, dass **char** genau 1 Byte, **short** mindestens 2 Bytes und **long** mindestens 4 Bytes breit ist. Somit kannst du dich auf folgende Reihen bezüglich der Größe verlassen:

**1 == char <= short <= int <= long**  
(<= bedeutet ist kleiner oder gleich)

## Gleitkommazahlen

Um es kurz zu machen, die Gleitkommatypen sind recht komplex, und auch hier lässt sich relativ schwer vorhersagen, wie breit diese auf deinem System sind. Häufig ist `float` mit 4 Bytes, `double` mit 8 Bytes und `long double` mit 10 oder gar 16 Bytes implementiert. Hierbei kannst du dich wiederum auch nur auf **folgende Reihenfolge** verlassen:

```
float <= double <= long double  
(<= bedeutet ist kleiner oder gleich)
```

Bitte ein Byte!  
Wie krieg ich jetzt  
die tatsächliche Größe  
für die Typen  
auf meinem Rechner  
heraus?

Die Byte-Größe der fundamentalen Typen auf deinem Rechner kannst du mit dem `sizeof`-Operator ermitteln. Einfach den Datentyp zwischen Klammern stellen, und der Operator liefert die **Byte-Größe** für den Typ zurück. Zum Beispiel kannst du dir sicher sein, dass folgende Ausgabe immer den Wert 1 zurückgibt:

```
cout << sizeof(char) << endl; *1
```

\*1 = 1 Byte  
(char ist immer 1)

Das ist ja gut und schön, aber ich wollte eigentlich nicht wissen, wie viele Bytes ein solcher Typ hat, sondern eher welchen Wert ich bspw. in einem `int` speichern kann oder wie viele Bits ein `char` tatsächlich auf meinem Rechner hat?

Die Antwort auf deine Frage findest du in der **Spezialisierung des Templates** (=Schablone) `numeric_limits` in `<limits>`. Gewöhnlich findest du dort für jeden fundamentalen Datentyp eine Spezialisierung. Der Großteil der Elemente in `<limits>` dient dazu, Gleitkommazahlen zu beschreiben.

Am meisten dürfte dich wohl interessieren, wie du den maximalen oder minimalen Wert eines fundamentalen Typs ermitteln kannst. Hierzu ein kurzer Überblick zu einigen Elementfunktionen, mit denen du bestimmte **Informationen zu Typen** in Erfahrung bringen kannst (**TYP** musst du natürlich durch deinen fundamentalen Typ ersetzen):

\*1 Hier bekommst du den **maximalen Wert** von **TYP** zurück.

\*2 Das gibt dir den **kleinsten Wert** von **TYP** zurück.

```
numeric_limits<TYP>::max() *1  
numeric_limits<TYP>::min() *2  
numeric_limits<TYP>::digits *3  
numeric_limits<TYP>::is_signed *4
```

\*4 Gibt **true** zurück, wenn dein **TYP ein Vorzeichen** hat. Ansonsten ist der Wert **false**.

\*3 Damit bekommst du die **Anzahl der Bits** von deinem **TYP** zurück.

## The Final Frontier

Da du jetzt weißt, dass die unendlichen Weiten deines Rechners **nicht** so **unendlich** sind, sollst du natürlich auch wieder etwas in die Praxis einsteigen und ermitteln, wie weit du auf deinem Rechner gehen kannst. **Lerne** deine Grenzen kennen!



[Einfache Aufgabe]

Finde heraus, wie bei dir der minimale und der maximale Wert von `int` lauten und wie viele Bits und Bytes ein `char` auf deinem Rechner hat.

[Schwierige Aufgabe]

Optional kannst du auch noch versuchen, herauszufinden, ob `char` mit oder ohne Vorzeichen eingebaut wurde.



Hier eine mögliche Musterlösung dazu ...

\*1 Hier bekommst du über die Elemente `max()` und `min()` den **maximalen** und **minimalen Wert** von `int` auf deinem System zurück. Diese Spezialisierung liegt in der Regel auch für alle anderen Typen vor.

\*2 Das gibt dir die **Anzahl der Bits** von einem `char` auf deinem Rechner zurück.

\*3 Mithilfe des `sizeof`-Operators kannst du den **Speicherverbrauch in Bytes** für einen Typen ermitteln. Im konkreten Fall wird `char` immer den Wert 1 zurückgeben (egal wie viele Bits 1 Byte auf deinem System hat).

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

```
int main()  
{  
    // Einfache Aufgabe:  
    cout << numeric_limits<int>::max() << endl; *1  
    cout << numeric_limits<int>::min() << endl; *1  
    cout << numeric_limits<char>::digits << endl; *2  
    cout << sizeof(char) << endl; *3
```



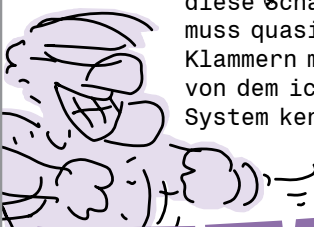
// Schwierige Aufgabe:

```
bool vorzeichen = numeric_limits<char>::is_signed; *4
if( vorzeichen == true ) { *5
    cout << "char ist als signed implementiert\n";
}
else { *5
    cout << "char ist als unsigned implementiert\n";
}
return 0;
}
```

\*4 Ist **char** mit einem **Vorzeichen** implementiert, gibt das Element **is\_signed** den Wert **true** zurück. Ansonsten wird **false** zurückgegeben. Den Rückgabewert speicherst du in **vorzeichen**.

\*5 Daher war die Aufgabe als **schwierig** gekennzeichnet, weil bis dato **if**-Überprüfungen noch nicht behandelt wurden. Hier wird praktisch überprüft, ob (**if**) **vorzeichen** gleich **true** ist. Ansonsten (**else**) ist **vorzeichen** gleich **false**. Es wird nur der entsprechende Anweisungsblock mit der Ausgabe ausgeführt, welcher wahrheitsgemäß zutrifft.

Ah, jetzt verstehe ich diese Schablonen auch. Ich muss quasi nur in den spitzigen Klammern meinen Typ reinsetzen, von dem ich die Grenze auf einem System kennenlernen will.



## Notiz

[Notiz]

Beachte bitte, dass es abhängig vom Typ und der Implementierung noch viel mehr Elementfunktionen in `<limits>` gibt, über die bestimmte Informationen ermittelt werden können. Gerade bei den Gleitkommatypen stehen enorm viele Informationen zur Verfügung.

Ich habe im INTERNET auf der Suche nach den Limits etwas über Makros in den Headerdateien `<climits>` für Ganzzahlen und `<cmath>` für Gleitkommatypen gelesen. Was ist damit?



Auf denen liegt mittlerweile ein Fluch. Weil es sich hierbei um **Relikte vergangener Zeiten** handelt, solltest du in C++ Programmen die Finger davon lassen. Ruhende Geister soll man nicht wecken. Nicht umsonst bietet dir schließlich C++ mit `<limits>` eine Spezialisierung in Form von Templates dafür an.

## Gut, dass es Grenzen gibt ...

Huh, das ist gar nicht so leicht, den Überblick über die verschiedenen Typen zu behalten. Dann kommt noch dazu, dass Eigenschaften wie Größe oder Wertebereiche von der Implementierung abhängen können.

Okay, Schrödinger, da ich deine Gedanken lesen kann, kriegst du hier noch einen Überblick über die fundamentalen Datentypen mit ihren „üblichen“ Wertebereichen und Größen an die Hand.

Zunächst die Ganzzahltypen, bei denen du auch gleich den Wahrheitswert und die Zeichentypen (die ja eigentlich auch Ganzzahltypen sind) vorfindest:

Typ	Speicher	Wertebereich
<code>bool</code>	1 Byte	0 oder 1 bzw. <code>false</code> oder <code>true</code>
<code>char</code>	1 Byte	-128 bis +127 bzw. 0 bis 255
<code>signed char</code>	1 Byte	-128 bis +127
<code>unsigned char</code>	1 Byte	0 bis 255
<code>wchar_t</code>	2 oder 4 Bytes	abhängig von der Implementierung
<code>short</code>	2 Bytes	-32.768 bis +32.767
<code>unsigned short</code>	2 Bytes	0 bis 65.535
<code>int</code>	4 Bytes	-2.147.483.648 bis +2.147.483.647
<code>unsigned int</code>	4 Bytes	0 bis 4.294.967.295
<code>long</code>	4 oder 8 Bytes	wie <code>int</code> oder -9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
<code>unsigned long</code>	4 oder 8 Bytes	wie <code>unsigned int</code> oder 0 bis 18.446.744.073.709.551.615

Du hast doch behauptet, dass `int` der natürlichste Typ zur BREITE DER PROZESSORARCHITEKTUR ist. Bei einem 16-Bit-Rechner hat `int` also 2 Bytes. Bei einem 32-BIT-RECHNER gibt mir der `sizeof`-Operator 4 Bytes zurück. Wie ist es aber jetzt bei den neuen 64-BIT-RECHNERN? Bei mir werden da nach wie vor 4 Bytes zurückgegeben. Ich hatte aber 8 Bytes gemäß deiner Aussage erwartet.



Okay, du hast es so gewollt! Das liegt am verwendeten Programmiermodell **P64**, wobei die Breite von **int** auf 32 Bits be-lassen wird und stattdessen der Typ **long** 64 Bits breit ist. Der Zeiger ist hierbei ebenfalls 64 Bits breit. Das Modell dürfte bei vielen Linux- und Unix-Systemen (also auch dem Mac) zu finden sein. 64-Bit-Windows verwendet hierbei oft das Modell **LLP64**, in dem sowohl **int** als auch **long** 32 Bits breit bleiben, aber die Zeiger auf Adressen 64 Bits lang sind (\*kurz Luft holen\*). Für einen 64-Bit-Typen nimmt man dann den frisch standardisierten Typen **long long**. Auf 32-Bit-Rechnern wie Windows, Mac, Linux wird das Modell **ILP32** verwendet, bei dem alles nur noch auf 32 Bits beschränkt ist. Das einzige Datenmodell, bei dem **int**, **long** und der Zeiger tatsächlich 64 Bits breit sind, ist im Augenblick **ILP64**, mit dem ich aber noch nicht das Vergnügen einer Bekanntschaft hatte.

[Zettel]

Zum besseren Verständnis. Das L steht für Long, das P für Pointer (Zeiger) und I für Integer. Die Zahlen 32 und 64 sprechen für sich.

Ergänzend zur bereits gezeigten Tabelle mit den Ganzzahltypen wurden mit dem neuen C++11-Standard (und z. T. schon lange davor) noch folgende Ganzzahl- bzw. Zeichentypen hinzugefügt:

Typ	Speicher	Wertebereich
char16_t	2 Bytes	abhängig von der Implementierung
char32_t	4 Bytes	abhängig von der Implementierung
long long	8 Bytes	-9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
unsigned long long	8 Bytes	0 bis 18.446.744.073.709.551.615

Jetzt fehlt dir nur noch der Überblick über die Gleitkommatypen:

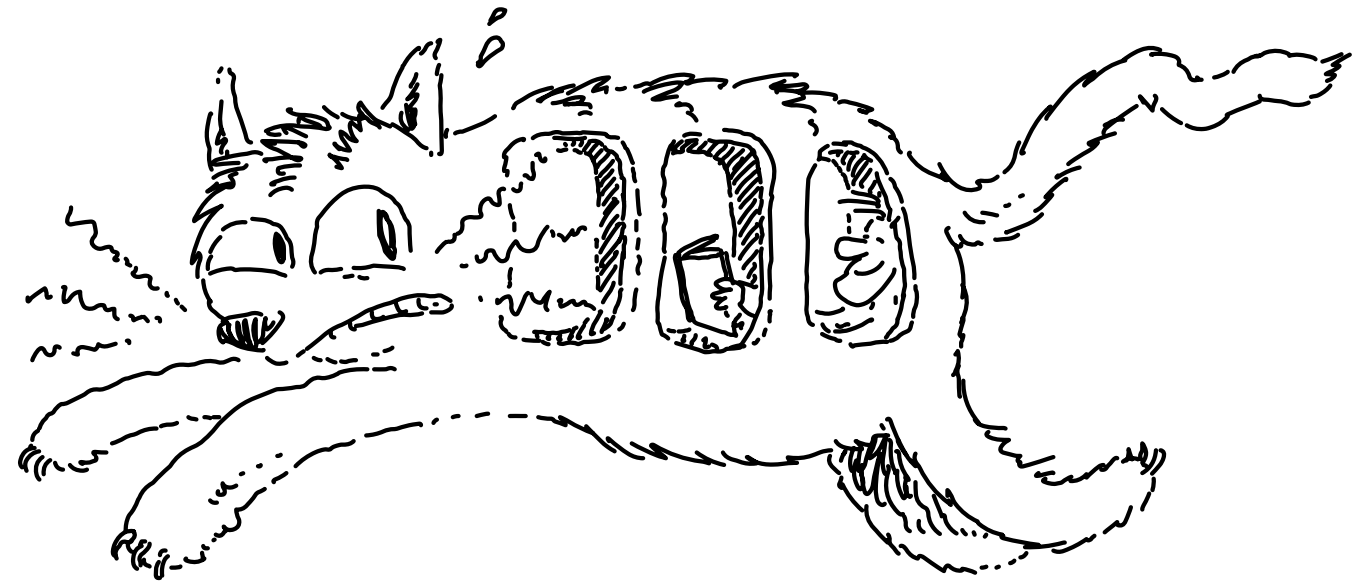
Typ	Speicher	Wertebereich
float	4 Bytes	1.2E-38 3.4E+38
double	8 Bytes	2.3E-308 1.7E+308
long double	10 (oder 16) Bytes	3.4E-4.932 1.1E+4.932

*Ich habe mal eben ein long long mit dem Wert 922337203685477580 probiert und mich mindestens dreimal vertippt. Gibt es da nichts, was es einfach macht, eine solche Riesenzahl einzugeben?*

Ja dazu gibt es jetzt etwas, Schrödinger! Aber es geht erst seit **C++14**, wo du jetzt einen digitalen Separator verwenden kannst. Einen solchen digitalen Separator kannst du mit einem einfachen Anführungszeichen oben (z. B. 1'000'000 für 1 Millionen) verwenden. Der Separator dient zur besseren Lesbarkeit. Hierzu ein paar Beispiele dazu:

```
int ival = 1'000'000;
long long llval =
9'223'372'036'854'775'807;
double dval = 0.000'351'2;
int gehtauch = 1'00'00'12'0;
int bval = 0b0010'1000'1100'0010;
```

SAYONARA



*... Mit dem Separator ...*

# INHALTSVERZEICHNIS

Vorwort .....	23
---------------	----

## Kapitel 1: Wir richten uns ein ...

### Entwicklungsumgebungen für C++

#### Seite 25

Brauche ich eine IDE zum Programmieren? .....	26	Lass uns endlich loslegen ... .....	31
Multikulturelle Sachen .....	27	Übersetzen mit einer Entwicklungsumgebung ... .....	31
Mikroweiche Sachen .....	28	g++ und clang++ .....	35
Die X-Sachen .....	29	... am Ende läuft es .....	37
Angebissene Äpfel .....	30		

## Kapitel 2: Elefanten können nicht fliegen, aber Schrödinger kann programmieren

### Erste Schritte in C++

#### Seite 39

Was ist eigentlich ein Computerprogramm? .....		Auf dem Weg zum Bildschirm .....	50
Ganz kurz und knapp für alle Schrödingers .....	40	Badewanne schon voll? .....	50
Die Sache mit dem main-Dings ... .....	41	Gib mir fünf .....	51
Unser erstes main-Dings soll laufen ... .....	43	Stream me up, Scotty .....	51
Endlich entspannen undträumen! .....	46	Gegenseitige Wahrnehmung ... .....	52
Kreuz und quer oder alles in Reih und Glied? .....	47	Manipulieren oder selber steuern? .....	53
Keinen Kommentar? .....	47	Noch ein wenig Brain-Streaming .....	54
Wie komme ich hier zum Bildschirm ...? .....	48		

## Kapitel 3: Verschiedene Typen für einen bestimmten Zweck

### Die C++-Basisdatentypen

#### Seite 57

Starke Typen .....	58	Das Pünktchen in der Werkstatt .....	68
Mein Name ist Schrödinger .....	59	Am Ende war das Pünktchen ... .....	69
Fundamentale und faule Typen .....	59	Zeichensalat .....	70
Deklaration und Definition .....	60	Doch ein ganzer Kerl? .....	71
Ganzer Kerl dank ... .....	61	Turmbau zu Babel .....	72
Zeichenfolgen von Ganzzahlen .....	62	Zum Flüchten ... .....	73
Positive und/oder negative Haltung und ein Endteil .....	62	Unicode-Unterstützung .....	73
Die Sache mit der Wahrheit ... .....	63	Zeichen für die Welt .....	74
Was nehmen wir für einen Typen? .....	65	Erste „Zeichen“ für eine Pause .....	76
Die Welt der ganzen Kerle .....	66	Auf die Größe kommt es an ... .....	78
Was für den einen das Komma, ist für den anderen der Punkt ... .....	67	Je größer, desto besser .....	79
		The Final Frontier .....	81
		Gut, dass es Grenzen gibt ... .....	83

## Kapitel 4: Von Zahlen verweht ...

### Arbeiten mit Zahlen

#### Seite 85

Manueller Rechenschieber .....	86	Keine Ahnung, aber ich verwende es trotzdem ....	101
Erweiterter Grundrechenzuweisungsoperator ...	87	Am Ende der Mathewelt .....	104
Futter für den Prozessor .....	88	Den Typ mag ich nicht .....	106
Kopfrechnen .....	90	Lass ihn doch ... .....	106
Achtung vor den Doppelgängern .....	90	Automatische Promotion .....	107
Nachsitzen in Mathe .....	92	Mit dem Hammer auf die Schraube .....	108
Wenn Grenzen überschritten werden .....	92	Warum man sich nicht auf JEDEN Typ einlassen sollte ... .....	110
Ungenaues Pünktchen .....	94	Der sanfte Typ .....	112
Schwächen offenlegen .....	95	Automatische Typenableitung .....	114
Mir raucht der Kopf .....	97		
Mathematische Spezialitäten .....	99		

# Kapitel 5: Eigene Entscheidungen treffen oder das Ganze nochmal bitte

## Kontrollstrukturen in C++

Seite 115

Endlich selbstständig sein und eigene Entscheidungen treffen .....	116	Immer diese Wiederholungen .....	134
Verzweigung, Abzweigung oder Kreuzung .....	119	Ein Schritt vor oder einer zurück ... ..	134
Wrong turn? .....	121	After a while ... ..	135
Ein ganz anderer Fall .....	125	Now for it! .....	136
Den Fall bearbeiten .....	127	Fußnoten nicht vergessen! .....	137
Den Fall analysieren .....	129	Nach oben oder nach unten .....	137
Also sprach Zarathustra .....	131	Und alles noch einmal ... ..	138

# Kapitel 6: Von gleichen und unterschiedlichen Typen, dem Sternchen und anderen ungemütlichen Sachen

## Arrays, Strings, Vektoren, Strukturen und Zeiger

Seite 141

Gleiche Typen in einer Reihe aufstellen .....	142	Endlich echte Schuhdaten .....	159
Die Fricke- und Fummelfraktion von Ze .....	143	Die gemischten Typen sind echt nützlich .....	162
Die Ze-Strings .....	144	Von Unionen, Aufzählungen und Synonymen ...	164
Gib mir rohen Input (Bio-Arrays) .....	145	1, 2, Freddy kommt vorbei,	
Das macht keinen Spaß .....	147	3, 4, schließe deine Tür ... ..	165
Krankheiten von Ze-Arrays und Ze-Strings .....	148	Die Lehre der Synonymie .....	169
Die gleichen Typen in einer Reihe aufstellen und wieder zurück .....	150	Leipziger Allerlei .....	170
Die Komfortklasse(n) von Ze++ .....	150	typedef Schrödinger hat_aller_kapiert_t .....	172
Reduzierter Bioanteil in vector .....	151	Weißt du, wie viele Sternlein am Himmel stehen?...	174
Reduzierter Bioanteil in string .....	152	Ich weiß, wo du wohnst ...! .....	175
Nie mehr Bio(-Arrays) .....	153	Einbruch in fremde Wohnungen .....	176
Am Ende hat es doch noch Spaß gemacht .....	155	Wohnorte ermitteln .....	176
Die Mischlinge .....	157	Sternenkunde .....	177
Zugriff auf die Mischlinge .....	158	Ze-Zeugs, Zeiger und wo soll das hinführen ...?! ..	178
		Zeiger auf nichts ...! .....	180

Wo geht's hier zur „Milky Way“? .....	181	Speicherhalde und Müllbeseitigung .....	184
Wo gibt's hier frischen RAM? .....	183	RAM mit WoW-Freunden auf Anfrage .....	186
Alles neu .....	183	RAM Unleashed .....	189

# Kapitel 7: Funktionen, das Ende von Copy & Paste ...

## Funktionen

Seite 193

Die Geister, die ich rufen will .....	194	Dinge, die man besser nicht beschwören sollte ... ..	212
Meine Hausgeister .....	195	Referenzen als Rückgabewert .....	214
Erster Kontakt zum Jenseits .....	197	Die Stille ist zerrissen .....	215
Geisterjäger im Wohnzimmer .....	199	Hausgeister zurückgeben ... ..	217
Opfergaben für die Geister .....	201	Jetzt bist du ein Medium .....	218
Als Kopie übergeben (Call-by-Value) .....	201	Spezielle Geister .....	220
Als Adresse übergeben (Call-by-Reference) .....	202	Werte für den Notfall ... ..	220
Referenzen als Funktionsparameter .....	203	Gleicher Name, unterschiedliche Typen .....	221
Ze-Zeugs als Funktionsparameter .....	204	Rollerblades für Funktionen? .....	221
Strukturen und Klassen als Parameter .....	205	main Programmende .....	223
Unsere Gaben wurden angenommen ... ..	206	Jenseits von Eden .....	224
Unendliche Stille .....	208	Am Ende der Geisterwelt .....	227
Das Jenseits antwortet dir .....	211		
Zeiger als Rückgabewert .....	212		

# Kapitel 8: Wie aus Chaos Ordnung entsteht

## Schlüsselwörter für Typen, Namensbereiche und die Präprozessor-Direktiven

Seite 229

Eigenschaften ohne Ende .....	230	Extrawurst-Gültigkeitsbereich .....	241
Klasse, die Speicherklasse .....	230	Einen neuen Lebensraum schaffen .....	242
Typqualifikationen .....	232	Betreten des neuen Lebensraumes .....	243
... und für die Funktionen auch noch		Using me .....	244
Extrawürstchen .....	233	Ein eigenes kleines Königreich .....	246
Mindesthaltbarkeitsdatum ändern .....	234	Anti-Globalisierung .....	249
Gleich-Gültigkeitsbereich .....	238	Anonymer Lebensraum .....	250

Lebensraum im Wohnzimmer .....	252	Die Zutaten für den leckeren Kuchen .....	266
Das #Ding vor dem Compiler .....	255	„Symbol(s) not found“, oder eine Zutat fehlt .....	267
#include „paste ohne copy“ .....	256	Die Einkaufsliste mit den Zutaten .....	268
#define „Symbol“ und „Makro“ .....	256	Ein nützliche Einkaufsliste, was alles so in einer Headerdatei verwendet werden könnte/sollte/muss ... .....	268
Die Übersetzung dirigieren .....	258	Die Zutaten vorbereiten und abwiegen .....	269
#Ich bestimme, was #du bekommst .....	259	... und jetzt alles in die Schüssel .....	270
„No such file or directory“, oder wo bin ich hier ... .....	260	Rein ins Vergnügen .....	271
Makros und Symbole, oder doch lieber nicht? .....	261	Meister der Quelldateien .....	275
#Ich h### all## v##sch###t .....	263		
Und jetzt alle zusammen! .....	265		

## Kapitel 9: Von Hexenmeistern, Todesrittern und Datenkapseln

### Klassen

#### Seite 277

Oben-ohne-Programmierung .....	278	Aufbauen und Vernichten .....	302
Klasse, Klassen! .....	279	Dienst nach Vorschrift .....	302
Objekte zum Mitnehmen .....	279	Wir übernehmen selbst ... .....	303
Ein Objekt erblickt das Licht der Welt .....	280	Konstruktoren mit mehreren Parametern .....	304
Kontrolle: Du kommst hier nicht durch .....	282	Konstruktoren effektiver initialisieren .....	305
Bei Klassendefinitionen den Überblick behalten .....	284	Klassenelemente direkt initialisieren .....	306
Tieferer Einblick in die Elementfunktionen der Datenkapsel .....	286	Am Ende alles wieder saubermachen ... .....	306
Du darfst hier nur lesen .....	288	Frühjahrsputz .....	307
Elementfunktionen voll im Einsatz .....	289	(K)ein Kartenhaus .....	310
Toll, diese Klassenfunktionen .....	291	Deep inside .....	312
Objekte erstellen .....	293	Spezielle Konstruktoren .....	312
Objekte auf die Welt bringen .....	294	Praxis Dr. Schrödinger .....	315
Zugriff auf die Öffentlichkeit der Klasse .....	295	Wohnung von Dr. Schrödinger .....	317
Indirekter Zugriff auf die Öffentlichkeit .....	295	The Big Three .....	318
Objekte verwenden .....	296	Spezielle Daten in der Kapsel .....	319
Die Geschichte von Objekten .....	299	Gute Freunde kann niemand trennen ... .....	321
		*Gong* Die letzte Runde wird eingeläutet .....	323
		Kampfanalyse .....	328

## Kapitel 10: Kino + WoW + Programmieren = viel Spaß

### Überladen von Operatoren

#### Seite 331

Eigene Rechengesetze .....	332	Weitere Operatoren überladen .....	351
Gestatten: operator .....	332	Logisch? Fast immer! .....	351
Gesetze für die Herren Operatoren .....	333	„Typenverbiegenumwandler“ überladen .....	351
Operatoren nur für Freunde .....	334	Input-Output-Kompott ... .....	352
Die Pärchen verwenden die Operatoren .....	335	Spezielle Operatorüberladungen in der Praxis .....	354
Mit vereinten Kräften .....	337	Spezialitäten auf dem Sofa .....	360
Glückliche Pärchen .....	340	Funktionsobjekte .....	360
Einsame Operatoren überladen .....	344	Indexoperator [] überladen .....	361
Das einsame Leben der einsamen Operatoren .....	346	new und delete überladen .....	361
Am Ende bleibt ein einsamer Operator .....	349		

## Kapitel 11: Schrödinger macht sein Testament

### Abgeleitete Klassen

#### Seite 363

Erben ohne Erbschaftssteuer .....	364	Wer bekommt was ... .....	392
Ewig schleichen die Erben .....	367	Keiner geht leer aus ... .....	394
Damit keiner leer ausgeht .....	371	Mehrfachvererbung .....	396
Jetzt das Kleingedruckte lesen .....	373	Mehrfachvererbung in der Praxis .....	398
Zugriffsrechte für den Beerbten .....	373	Lohnt sich die Mehrfachvererbung überhaupt? .....	401
News: Konstruktoren vererben (C++11) .....	376	Virtuelles Vererben .....	403
Das Kleingedruckte in der Praxis .....	377	Virtuelle Teilchen verwenden .....	405
So macht erben langsam Spaß .....	380	Zwischen Virtualität und Realität .....	406
Private Mitglieder durchreichen ... .....	381	Abstrakte Welten .....	408
Erbe verwenden und erweitern .....	382	Ein alter Bekannter ... .....	408
Redefinition .....	383	Abstrakte Vielgestaltigkeit .....	410
Use me ... .....	383	Jetzt wird es vielseitig – Polymorphie .....	411
Unser Anlageberater verwaltet das Erbe .....	385	Virtuelle Zerstörungskraft .....	413
Ordentlich angelegt .....	387	Was vom Abstrakten übrig blieb .....	414
Konstruktives und destruktives Vererben .....	389	Was bist du denn? .....	415
Implizite Klassenumwandlung .....	391	override und final .....	416



# Kapitel 12: Ausstechformen für die Plätzchen

## Templates

Seite 419

Funktionen zum Ausstechen .....	420	Klassen-Ausstecher-Elementfunktion definieren ..	430
Verschiedene Teigsorten .....	422	Klassen-Ausstecher-Elementfunktion überschreiben	431
Plätzchen backen .....	424	Objekte ausstechen .....	432
Am Ende sind nur noch Krümel da ... ..	428	Klassen-Ausstecher in der Praxis .....	433
Klassen zum Ausstechen .....	429	Klassen-Ausstecher in der Wohnung .....	438

# Kapitel 13: Der Schleudersitz für den Notfall

## Ausnahmebehandlung

Seite 441

Versuchen, werfen und auffangen .....	442	Wir probieren es aus .....	464
Noch ein paar Hinweise für das Werfen .....	444	Logischer Fehler: out_of_range .....	464
Jetzt schmeiß schon! .....	446	Logischer Fehler: invalid_argument .....	465
Was passiert danach ...? .....	447	Logischer Fehler: length_error .....	466
Homerun .....	450	Logischer Fehler: ios_base::failure .....	467
Mit Klassen um sich schmeißen .....	453	Standardausnahme-Kontrolle .....	468
Nix wie weg hier .....	453	Ausnahme-Spezifikation und noexcept? .....	470
Schmeißen mit ganzen Klassen .....	455	noexcept .....	470
Homerun mit Klassen .....	458	Hasta la vista, baby .....	471
(Standard-)Ausnahmen im Angebot .....	461	Ausnahmen verweigern .....	472
What ist dein Problem ...? .....	462	Keine Erwartungen .....	474
Ausnahmen im System .....	463		

# Kapitel 14: Unterwäsche, 100 % Baumwolle, Doppelripp

## Die Standardklasse string

Seite 475

Schurz, Schürzen, Schürzenjäger .....	476	Wie groß isses denn? .....	480
Strings anlegen und zuweisen .....	476	Rohe Strings .....	480
Zugriff auf die einzelnen Zeichen .....	479	Noch mehr Unterwäsche ... ..	481

Und noch mehr davon .....	483	Überladene Operatoren und Ein-/Ausgabe .....	488
Klamottenkiste .....	485	Ich kann's nicht mehr hören: Strings .....	490
String konvertieren und manipulieren .....	485	Alles sauber dank „Schwarzer Zwerg“ .....	494
Such! .....	486		

# Kapitel 15: Ströme ohne Isolierkabel verwenden

## Der Umgang mit Streams und Dateien

Seite 495

Gib aus den Strom .....	496	Datei-Slang .....	526
Rohe Eier raus ... ..	498	Vorhandene Ströme für Dateien .....	527
Mehr formatierter Output bitte .....	499	Strom für die Datei anschließen .....	527
Wir schwenken die Flagge ... ..	500	Plug-ins für den Dateistrom .....	528
Jetzt ist es raus ... ..	502	Den Dateistecker ziehen .....	530
Wir ändern die Richtung .....	505	Alles gesichert ...? .....	531
Rohe Eier rein ... ..	505	Sauber lesbare Sache .....	531
Wo ist mein Input? .....	507	Stück für Stück .....	532
Jetzt ist es drin ... ..	511	Zeile für Zeile .....	534
Wir manipulieren die Ströme .....	513	Ganze Happen ... ..	535
Manipulation ist alles ... ..	516	Wahlfreier Zugriff .....	536
Ordentlich manipuliert ... ..	520	Daten wiederherstellen .....	537
Auch ein Strom hat seine Fehler .....	522	Ein Strom für einen String? .....	541
Erst den Strom abstellen ... ..	524	Schürzenjäger-Strom .....	543
Die Sicherung ist geflogen .....	525	Ohne Isolation .....	545
Kann man auch was speichern? .....	526		

# Kapitel 16: Ausstechformen für Faule

## Einführung in die Standard Template Library (STL)

Seite 547

Fertigkuchen von Dr. STL ...? .....	548	Detaillierteres Arbeiten mit sequenziellen	
Verschiedene Behälter (Container) .....	549	Fertigkuchen .....	558
Algorithmen und Iteratoren .....	551	Behälter erstellen .....	558
Besser als „Selbermachen“ .....	552	Zutaten hinzufügen .....	559
... und schmeckt auch noch gut! .....	555	Zugriff auf den Teig .....	560

Wie viel passt rein, und wie viel ist drin ...?	561	Funktionsobjekte	581
Raus damit ...!	561	Kategorie von Iteratoren	583
Tausch mit mir, oder gib mir alle	562	Iterator-Adapter	585
Mixen, sortieren und rühren	562	Die Hilfsmittel für Fertigmöbel und	
Sequenzielle Fertigmöbel abschmecken	563	Zutaten im Einsatz	586
Bereit zum Essen ...	568	Hilfe für den Iterator	588
Detaillierteres Arbeiten mit assoziativen		Allmählich wird es öde ...	589
Fertigmöbel	570	Die fleißigen Arbeiter	592
set und multiset	571	Nicht-modifizierende Algorithmen	592
map und multimap	572	Modifizierende Algorithmen	593
Bitte ein Bit-Feld ...!	572	Löschende Algorithmen	594
Assoziative Möbel backen	573	Mutierende Algorithmen	595
(multi)set me up, baby!	573	Sortierende Algorithmen	596
Now (multi)map me!	575	Algorithmen für sortierte Bereiche	597
Bitte ein Bit!	576	Algorithmen verwenden	598
Auch assoziative Möbel kann man essen	579	Ende gut, alles gut ...	602
Zwischen Fertigmöbel und weiteren Zutaten	581		

## Kapitel 17: Schöne neue Welt C++11

### C++11 – der neue Standard

Seite 605

C++ auf dem neuesten Stand(ard)	606	Cool, das neue Zeug	616
auto/decltype	607	Weitere nützliche Features	617
Einfachere Initialisierung	607	Noch mehr Neuigkeiten ...	619
Lambda-Funktionen	608	Ein neues Array?	619
Range-based-loop	608	Eine neue verkettete Liste?	620
Explizite delete- und default-Funktionen	609	Hasch? Ist das nicht illegal?!	620
nullptr	609	Neue Algorithmen	621
constexpr	610	Tuple? Tulpe?	622
Ein Konstruktor ruft einen anderen		Neue Planeten braucht das Universum	623
Konstruktor auf	611	Neue Backmischungen sind auch gut	626
Move your body	611	Kluge Zeiger	628
Neues Zeug im Einsatz	613	Ich bin der Klügste hier (shared_ptr)	630
auto/decltype	613	Schwacher Zeiger (weak_ptr)	631
{}-Initialisierer verwenden	613	Egoistischer, aber kluger Zeiger (unique_ptr)	633
Lambda-Funktion	614	Klug auch in der Praxis	634
Move my own class	615	Bist du auch so klug ...?	637

Von Hieroglyphen und regulären Ausdrücken	639	Wir nehmen jetzt die Fäden in die Hand	658
Mini-Sprachkurs zu den Hieroglyphen	639	Nur nicht den Faden verlieren	661
Objekt für die Hieroglyphen	643	Schütze deine Daten	663
Die Algorithmen für Hieroglyphen	643	Ein Schloss für den Mutex	666
Suchergebnis analysieren	644	Sicheres Initialisieren	667
Suchen mit Hieroglyphen	645	Totgesperrt	668
Cleopatra ist da ...	652	Einmal bitte ...	670
Parallele Welten	654	Am Ende des Fadens ...	672
Viele Fäden erzeugen	655	„Konditions“-Variable ...?	672
Bist du jetzt ein Faden oder nicht?	657	Zusammenfassung	674
Argumente für den Thread	657		

## Kapitel 18: C++ 14 – der Neue!

### C++14 – der allerneueste Standard

Seite 675

Schon wieder ein neuer Standard?	676	Und dann noch etwas für die	
Der Compiler weiß es doch sowieso		Zahlenzerstückler	680
immer besser	677	Mr. Holmes, bitte übernehmen Sie ...	681
Dann mach es doch auch selbst bei		Der Tanz mit den Lambda-Parametern	682
den $\lambda$ -Funktionen	677	Alte Sachen aufheben oder ausmisten?	683
Gammelcode an den Compiler verraten?	678	Mir reicht es jetzt mit der neuen Welt	684
Etwas für die Bitverdrehler unter uns	679	Noch ein paar mehr C++14-Sachen	685

Index	689
-------	-----

# INDEX

## Symbole

-> 295  
.  
295  
\\ 196, 243, 287  
& 203  
&& 611  
# 255  
<< 54  
überladen 352, 356  
>> 54  
überladen 352, 356  
~ 306  
\\a 73  
<array> 619  
\\b 73  
<condition\_variable> 672  
<cstdlib> 45  
\_\_DATE\_\_ 264  
\\ddd 73  
#define 256, 261  
[[deprecated]] 678  
#elif 258  
#else 258  
#endif 258, 259  
(ENTER) 161  
#error 264  
\\f 73  
\_\_FILE\_\_ 264  
#if 258  
#ifdef 258  
#ifndef 258, 259  
#include 256, 259, 267  
<limits> 82  
#line 264  
\_\_LINE\_\_ 264  
<memory> 628  
<mutex> 660, 663  
\\n 73

#pragma 264  
<regex> 643  
<string> 476  
<thread> 655  
\_\_TIME\_\_ 264  
--, überladen 346  
!, überladen 349  
!=, überladen 351, 354  
[], überladen 361  
+, überladen 338  
++, überladen 346  
+=, überladen 340  
==, überladen 351, 354  
#undef 257  
<unordered\_map> 620  
<unordered\_set> 620  
\\uXXXX 73  
\\v 73  
\\xhh 73

## A

Abbruchbedingung 135  
Abgeleitete Klassen 364  
Ableitung 371  
abort 223  
abort() 471  
abs() 99  
Abstrakte Klasse 408  
Adapterklasse 549  
Addition (+) 58  
adjacent\_find() 593  
Adressoperator 177, 182  
advance() 588  
Akustisches Signal 73  
Algorithmus 551, 581, 592  
C++11 621  
für sortierte Bereiche 597  
löschender 594

modifizierender 593  
mutierender 595  
nicht-modifizierender 592  
sortierender 596

Aliase Templates 440  
all\_of() 621  
AND → Logisches UND  
Anführungszeichen 76  
Anweisungsblock 41, 195  
any() 577  
any\_of() 621  
arg() 99  
Argument 106  
Arithmetische Berechnungen 101  
Arithmetischen Operationen 108  
Arithmetische Operatoren 86  
Array 619  
ASCII-Code 70  
ASCII-Tabelle 74  
ASCII-Zeichensatz 76  
assign() 561  
async() 673  
at() 560  
Ausgabe formatieren 499  
Ausgabe, unformatierte 498  
Ausnahmebehandlung 441  
Ausnahmefehler 223  
Ausnahme-Handler 444  
auto 232, 239, 607, 613, 677  
auto\_ptr 628

## B

back() 560  
bad() 523  
bad\_alloc 463  
bad\_cast 463  
bad\_exception 463  
bad\_typeid 463

basic\_ 496  
Basisklasse 365  
Basistypen 58  
Bedingte Kompilierung 258, 263  
Bedingungsvariable 672  
Behälter 549  
    *assoziativer* 570  
    *sequenzieller* 558  
Bezeichner 59  
    *eindeutig, gültig* 66  
Bidirektionaler Iterator 584  
Binär 90  
Binärcode 40, 70  
Binäre Operatoren überladen 332  
binary\_search() 597  
Bits 79  
bitset 550, 572  
bool 79, 83  
boolalpha 514  
Boolescher Ausdruck 132  
Boolescher Typ 63  
boost-Bibliothek 104  
break 126, 128  
Buffer-Overflow 145, 155  
Byte 79

## C

C++14 675  
C++17 676  
Call-by-Reference 202  
Call-by-Value 201  
call\_once 670  
capacity() 561  
C-Array 619  
case 126, 128  
Case sensitivity 42  
Cast-Operator überladen 351, 356  
catch 444  
catch( ... ) 444  
cerr 50, 51  
char 70, 71, 75, 76, 79, 81, 83, 125  
char\* 175

char16\_t 73, 84  
char32\_t 73, 84  
cin 51, 52, 55  
class 163, 279, 421, 429  
clear() 523, 561, 571  
clog 50, 51  
close() 530  
Codepage 850 45  
condition\_variable 672  
conj() 99  
const 208, 232, 240, 261, 319  
const\_cast<TYP> 109  
constexpr 610, 670  
Container → Behälter  
copy() 594  
copy\_backward() 594  
copy\_n() 621  
count() 571, 593  
count\_if() 593  
cout 50, 51, 53, 55, 75  
c\_str() 537

## D

Data Race 664  
Datei 526  
    *lesen* 531  
    *öffnen* 527  
    *Position* 527, 536  
    *Puffer* 527  
    *schließen* 530  
    *schreiben* 531  
Datenkapsel 278  
Daten, Klassen 281, 319  
Daten speichern 58  
Datentyp, abstrakter 278  
Deadlock 668  
dec 501, 514  
decltype 607, 613  
default 128, 609  
default-Marke 127  
Default-Parameter 220  
Definition 60

Deklaration 60, 66, 268  
Dekrement 134  
Dekrementoperator 134  
delete 185, 609  
    *überladen* 361  
deque 549, 558  
Destruktor 306, 307, 318  
    *Vererbung* 389  
    *virtual* 413  
detach() 656  
Dezimale Schreibweise 62  
Diakritische Zeichen 70  
Direktive 255  
distance() 588  
divides<T> 583  
domain\_error 461, 466  
Doppelte Genauigkeit 98  
Dos-Latin.1 45  
double 67, 84, 98, 108, 184  
do-while-Schleife 131  
dynamic\_cast 463  
dynamic\_cast<TYP> 109  
dZeiger 185

## E

Einfache Genauigkeit 98  
Einlesen, unformatiert 505  
Einzelne Anführungszeichen 71  
Elementfunktion 153, 155,  
281, 286  
    *const* 288  
    *Klassen-Template* 430  
    *Spezialisierung* 431  
    *überschreiben* 431  
Elementinitialisierer 305, 392  
empty() 561  
endl 52, 53, 515  
ends 515  
eof() 523  
EOF 525  
equal() 593  
equal\_range() 597

equal\_to<T> 583  
erase() 561, 571  
Erweiterte Genauigkeit 98  
exception 461  
Exception-Handler 444  
exception handling 442  
exit 223  
explicit 314, 316  
explizit 233  
Explizite Typumwandlung 108  
Extern 230, 234, 238

## F

fail() 523  
Fehlerklasse 453  
Fehlermeldung 47  
Fehlerüberprüfung 187  
fill() 499, 594  
fill\_n() 594  
final 372  
find() 571, 592  
find\_end() 592  
find\_first\_of() 593  
find\_if() 592  
find\_if\_not() 621  
fixed 501, 514  
flags() 500  
flip() 572  
float 84, 98  
floating point 67  
flush 515  
for\_each() 582, 592  
for, Range-based 608  
for-Schleife 131  
Forward-Iterator 584  
forward\_list 620  
friend 322, 330  
    *Operatoren überladen* 334  
front() 560  
fstream 527  
Fundamentale Typen 66

Funktion 106, 194  
    *beenden* 223  
    *Definition* 197  
    *Deklaration* 197  
    *Elementfunktion* 194  
    *inline* 222  
    *mit Parameter* 201  
    *Rückgabewert* 211  
    *Standardparameter* 220  
Funktionsaufruf 198  
Funktionsobjekt 360, 517, 581  
    *vordefiniertes* 582  
Funktionsoperator, überladen 360  
Funktionsparameter  
    *als C-Array* 204  
    *als C-String* 204  
    *als Kopie* 201  
    *als Referenz* 203, 206  
    *als Zeiger* 202  
    *const* 208  
    *Klassen* 205  
    *Strukturen* 205  
Funktionsrückgabewert 211  
    *als Referenz* 214  
    *als Zeiger* 212  
    *lokale Daten (!)* 212  
    *zwei Werte* 218  
Funktions-Template 420, 543  
Funktionsüberladung 221,  
225, 227  
Funktort 517, 581

## G

Ganzzahlen 62, 65, 76, 79, 94, 134  
Ganzzahltypen 59, 61, 83  
generate() 594, 599  
generate\_n() 594  
get() 505  
get\_id() 657  
getline() 489, 506  
get<N>(bezeichner) 622  
Gleitkommatypen 59, 67, 68, 84

Gleitkommazahlen 80, 94, 125  
Gleitpunkttypen 67  
good() 523  
greater\_equal<T> 583  
greater<T> 583  
Groß- und Kleinschreibung 59, 66  
Grundlegende Typen 59  
Grundrechenzuweisungs-  
operator 87  
Gültige Werte 61  
Gültigkeitsbereich 195, 199,  
238, 245  
    *Destruktor* 308

## H

Hashtabelle 620  
Headerdatei 43, 99, 266, 268  
    *<iostream>* 48  
hex 501, 514  
Hexadezimale Schreibweise 62  
Hexadezimale Werte 45  
Hexadezimale Zahl 73  
Hilfszeiger 190

## I

if 82, 125  
if-Anweisung 117  
if-Anweisungsblock 117  
ifstream 527  
ignore() 161, 509  
ILP32 84  
ILP64 84  
Implementierung 68  
Implizite Konvertierung 110  
Indexoperator[] 143  
Indexoperator, überladen 361  
Indirektionsoperator 176  
Initialisieren 61, 67  
Initialisierung der Schleifen-  
variablen 132  
Initialisierungsliste 305, 309, 319

Inkrement 134  
 Inkrementoperator 134  
 inline 222, 262, 287, 425, 430  
 Input-Iterator 583  
 insert() 559, 571  
 Insert-Iterator 585  
 Instanz 279, 294, 299  
 int 61, 66, 81, 83, 108, 125  
 Integer 61  
 Integrale Ausweitung 111  
 internal 501, 514  
 invalid\_argument 461, 465  
 iomanip 515  
 ios::  
   ios::app 529  
   ios::ate 529  
   ios::badbit 523  
   ios::beg 536  
   ios::binary 529  
   ios::cur 536  
   ios::end 536  
   ios::eofbit 523  
   ios::failbit 523  
   ios::fmtflags 500  
   ios::goodbit 523  
   ios::in 529  
   ios::openmode 529  
   ios::out 529  
   ios::trunc 529  
 ios\_base 496  
   ios\_base::failure 462, 467  
 iostate 522  
 ostream 43  
 ostream-Bibliothek 48, 49  
 iota() 621  
 ISO-Latin-1 72  
 is\_open() 530  
 IST-Beziehung 365  
 istream 505  
   istream::getline() 511  
 istringstream 541  
 Iterationen 131  
 Iterator 551  
   Adapter 585  
   Kategorien 583  
 iter\_swap() 588, 594

## J

join() 655  
 joinable() 657

## K

Klasse 279  
   abstrakte 408  
   Ausnahmen 453  
   Daten 280  
     dynamische 320  
     konstante 319  
     statische 321  
   Fehler 453  
   Objekte 299  
   Zugriffsrechte 373  
 Klassendefinition 280, 281, 284  
 Klassen-Template 429  
   Elementfunktionen 430  
   Objekte erzeugen 432  
 Klassische Konvertierung 111  
 Kleiner 120  
 Komma 60  
 Kommazahlen 134  
 Komplexe Zahl 101  
 Konstruktor 302  
   mit Parameter 304  
   Vererbung 389  
 Konvertierungskonstruktor 314, 316, 317  
 Kopie, flache 324  
 Kopierkonstruktor 312, 316, 317, 318, 320, 324, 325  
 Kopie, tiefe 325  
 Korrekte Konvertierung 111

## L

Lambda-Funktion 591, 608  
 Laufzeit 183  
 Leerer Vektor 159  
 left 501, 514  
 length\_error 461, 466  
 less\_equal<T> 583  
 less<T> 583  
 list 549, 558  
 LLP64 84  
 lock() 669  
 Lock 666  
 lock\_guard 666  
 logical\_and<T> 583  
 logical\_not<T> 583  
 logical\_or<T> 583  
 logic\_error 461  
 Logischer Operator 122  
 Logisches NICHT 122  
 Logisches ODER 122  
 Logisches UND 122  
 long 61, 83, 125  
 long\* 175  
 long double 84, 98  
 long (int) 66  
 long int 61, 63  
 long long 84  
 lower\_bound() 597

## M

Mac 44  
 main() 41, 42, 223  
 Main() 46  
 main-Funktion 159  
 make\_heap() 596  
 make\_tuple() 622  
 Makro 256, 261  
 Manipulator 52, 513  
   benutzerdefinierter 516  
   mit Parameter 517  
   ohne Parameter 516

map 550, 570  
 Maschinencode 40  
 max() 81  
 max\_size() 561  
 Mehrfachvererbung 396  
   virtual 403  
 Memory Leaks 217  
 merge() 562, 597  
 min() 81  
 min\_element() 593  
 minus<T> 583  
 mismatch() 593  
 Modul 265  
 modules<T> 583  
 Modulo-Operator 86  
 move() 562, 612, 621  
 move\_backward() 621  
 Move-Semantik 611, 615  
 MS Windows 44  
 multimap 550, 570  
 multiplies<T> 583  
 multiset 550, 570  
 mutex 660, 663

## N

Namensbereich 242  
 Namenskonflikt 249  
 Namensraum 241, 246, 252  
   anonym 250  
   std 43  
 namespace 241, 242  
 negate<T> 583  
 new 463  
   überladen 361  
 next\_permutation() 595  
 noboolalpha 514  
 noexcept 470  
 none\_of() 621  
 norm() 99  
 noshowbase 514  
 noshowpoint 514

noshowpos 514  
 noskipws 514  
 No such file or directory 271  
 NOT → Logisches NICHT  
 not\_equal\_to<T> 583  
 notify\_one() 673  
 noutitbuf 514  
 nouppercase 514  
 nt\_element() 596  
 nullptr 607, 609

## O

Objekt 278, 293, 296  
   const 319  
   erzeugen 294  
   Klassen-Template 432  
 Objektzeiger, überladen 359  
 oct 501, 514  
 ofstream 527  
 Oktale Schreibweise 62  
 Oktale Zahl 73  
 once\_flag 670  
 OOP 278  
 open() 528  
 operator 332  
 operator() 517  
 Operator 80, 86, 90, 120  
   gleich 120  
   größer 120  
   größer oder gleich 120  
   kleiner 120  
   kleiner oder gleich 120  
   nicht überladbar 333  
   überladbar 333  
   überladen 332  
     friend-Funktion 334  
     Vererbung 375  
   ungleich 120  
 Operatoren 54  
 OR → Logisches ODER  
 ostream 497

ostreamstream 541  
 out\_of\_range 461, 464, 479  
 Output-Iterator 584  
 overflow\_error 462

## P

pair<> 572  
 partial\_sort() 596  
 partial\_sort\_copy() 596  
 partition() 595  
 Pfeiloperator, Klassen 295  
 plus<T> 583  
 polar() 99  
 Polymorphie 411, 416  
 pop() 561  
 pop\_back() 561  
 pop\_front() 561  
 pop\_heap() 596  
 Postfix-Schreibweise 134  
 Prädikat 583, 590  
 Präfix-Schreibweise 134  
 Präprozessor 43, 255  
 precision() 499  
 prev\_permutation() 595  
 Priorität der Klammerung 95  
 priority\_queue 550, 560  
 private 282, 285, 373, 377  
   Mitglieder durchreichen 381  
 Programmdatei, ausführbar 40  
 Programmende 223  
 protected 373, 379  
 Prozessorarchitektur 83  
 public 282, 285, 373, 378  
 Puffer 50  
 Pufferüberlauf 147, 149  
 Pufferunterlauf 147  
 Punkteoperator 158  
   Klassen 295  
 Punkt-vor-Strichrechnung 87  
 push() 560  
 push\_back() 559

push\_front() 559  
push\_heap() 596  
put() 506  
putback() 506

## Q

Quelldatei 266, 269  
queue 550, 560

## R

Race Condition 664  
rand() 599  
Random-Access-Iterator 584  
random\_shuffle() 595  
Range-based-loop 608  
range\_error 462  
Raw-String-Literale 480  
rbegin() 585  
rdstate() 523  
recursive\_mutex 663  
recursive\_timed\_mutex 663  
Redefinition 383  
Referenz 203  
    zyklische 631  
Regel der Großen Drei 318  
regex 643  
regex\_error 462  
regex\_match 643, 645  
regex\_replace 643, 649  
regex\_search 643, 646  
Regulärer Ausdruck 639  
reinterpret\_cast<TYP> 109  
Rekursion 386  
remove() 567, 595  
remove\_copy() 595  
remove\_copy\_if() 595  
remove\_if() 595  
rend() 585  
replace() 594  
replace\_copy() 594  
replace\_copy\_if() 594  
replace\_if() 594

reserve() 561  
reset() 572  
resetiosflags() 515  
resize() 561  
return 106, 211, 223  
reverse() 562, 567, 595  
reverse\_copy() 595  
Reverse-Iterator 585  
RGB 170  
right 501, 514  
rotate() 595  
rotate\_copy() 595  
Rückgabetyt 106  
Rückschritt/Backspace 73  
Rundungsfehler 94  
runtime\_error 461  
Rvalue-Referenz 611

## S

Schleifen 131  
    Abbruchbedingung 132  
    Variable 135  
scientific 501, 514  
search() 593  
search\_n() 593  
seekg() 536  
seekp() 536  
Seitenvorschub 73  
Semikolon 60, 133  
set 550, 570  
set() 572  
setbase() 515  
set\_difference() 597  
setfill() 515  
set\_intersection() 597  
setiosflags() 515  
setprecision() 515  
setstate() 523  
set\_symetric\_differnce() 597  
set\_terminate() 471  
set\_union() 597  
setw() 515

shared\_ptr 628, 630  
short 61, 83  
short (int) 66  
showbase 501, 514  
showpoint 501, 514  
showpos 501, 514  
signed 66, 71, 113  
signed char 83  
size() 151, 561  
sizeof 80, 81, 175  
skipws 514  
Smart Pointer 628  
smatch 644  
sort() 562, 596  
sort\_heap() 596  
Spagetti-Code 136  
Speicherklasse 230  
Speicherlecks 148, 192  
Speichern von Daten 58  
Speicherobjekt 60  
Speicherverbrauch in Bytes 81  
Speicherverwaltung 183  
Spezialisierung, Klassen-  
Template 431  
splice() 562  
sregex\_iterator 648  
sregex\_token\_iterator 651  
sstream 541  
stable\_partition() 595  
stable\_sort() 596  
stack 550, 560  
Stack-Unwinding 453  
Standardfehlerklassen 461  
Standardkonstruktor 302, 325  
Standardkopierkonstruktor 312  
Standardparameter 220, 224, 227  
Standardstrom 497  
static 231, 236, 238, 321  
static\_cast<TYP> 108  
std 55  
std:: 244, 249  
stdio 501  
Sternchen 174, 182

STL 429, 476, 548  
Stream 48  
Stream-Iterator 585  
string 144, 150, 154, 476  
    + 488  
    += 488  
    << 488  
    == 488  
    >> 488  
    append() 485  
    at() 479  
    capacity() 480  
    clear() 480  
    copy() 485  
    C-String 485  
    data() 485  
    empty() 480  
    erase() 485  
    find() 486  
    find\_first\_of() 486  
    getline() 489  
    Indexoperator 479  
    insert() 485  
    Länge ermitteln 480  
    length() 480  
    max\_size() 480  
    npos 486  
    Operatoren 488  
    replace() 485  
    resize() 480  
    rfind() 486  
    size() 480  
    stringstream 541  
    Strom 541  
    Suche 486  
    verketten 488  
    Zuweisungsoperator 477  
string::getline() 511  
Strom 496  
    Ausgabe  
        formatierte 499  
        unformatierte 498  
    Dateien 526

Eingabe 505  
Manipulator 513  
Standard- 497  
String 541  
Stroustrup, Bjarne 318  
struct 163, 164  
Strukturzeiger 187  
Suchen, string 486  
swap() 562  
swap\_ranges() 594  
switch 125, 126, 128  
Symbolische Konstante 76  
system() 45  
system\_error 462

## T

Tabulator, horizontal 73  
Tabulator, vertikal 73  
tellg() 536  
tello() 536  
template 421, 429  
template<> 425  
Template, Funktionen 420  
Template, Klassen 429  
Temporäre Struktur 160  
terminate() 444  
Terminierungszeichen 53  
this-Zeiger 297, 300  
thread 655  
thread\_local 667  
throw 444, 448  
timed\_mutex 663  
to\_long() 572  
top() 560  
to\_string() 572  
transform() 594, 599  
true → Wahr  
try 447  
tuple 622  
Turmbau zu Babel 72  
typedef 172, 434  
typeid 415, 427, 463

typeid 415, 427  
Typen  
    Basistypen 58  
    komplexere 66  
typename 421, 429  
Typnamen vereinfachen 169  
Typqualifikation 232  
    const 232  
Typumwandlung, Klassen 391

## U

Überladen  
    Operatoren 332  
    binäre 332  
    unäre 344  
Überlaufender Wert 110  
Ubuntu-Linux 44  
Unär 90  
Unäre Operatoren überladen 344  
Undefined symbols 271  
underflow\_error 462  
unget() 506  
Unicode 73, 74  
union 164, 172  
unique() 595  
unique\_copy() 595  
unique\_lock 666, 669  
unique\_ptr 628, 633  
unitbuf 501, 514  
unordered\_map 620  
unordered\_multimap 620  
unordered\_multiset 620  
unordered\_set 620  
unsetf() 500  
unsigned 62, 63, 71, 113  
unsigned char 83  
unsigned int 83, 108, 143  
unsigned long 83  
unsigned long long 84  
unsigned short 83  
Unterstrichzeichen 59  
unwahr 63

upper\_bound() 597  
uppercase 501, 514  
using 241, 244  
using namespace std 249  
US-Schreibweise 67  
UTF-8-Kodierung 72, 73  
UTF-8-Literale 45

## V

vector 150, 151, 154, 299, 549, 558  
Vererbung 364, 371  
    *Destruktoren* 389  
    *Elementinitialisierer* 392  
    *erweitern* 382  
    *Konstruktoren* 389  
    *Mehrfachvererbung* 396  
    *Operatorüberladung* 375  
    *Redefinition* 383  
    *Zugriff auf Mitglieder* 383  
Vergleichsoperatoren 120  
Vielgestaltigkeit 411

virtual 233, 404, 408  
    *Destruktor* 413  
Virtuelle Vererbung 403  
void 60  
volatile 233  
Vorzeichen 65, 66, 82  
vorzeichenlos 62

## W

Wahr 63  
wait() 673  
wcerr 76  
wchar\_t 75, 79, 83  
wcin 76  
wcout 75, 76  
weak\_ptr 628, 631  
Weihwasser 60  
Weisheit 60  
Werte, negativ 62  
Werte, positiv 62  
what() 462  
while-Schleife 131

width() 499  
Wikipedia 68  
WoW 58  
ws 515

## Z

Zeichenkette 144  
Zeichenerliterale 71  
Zeichentypen 70  
Zeiger 178  
Zeilenende 73  
Zielbereich des Puffers 148  
Zufallszahlen 237  
Zugriffsoperator 196, 243,  
248, 287  
Zugriffsrechte 373  
Zugriff, wahlfreier 536  
Zuweisung 61, 106  
    (=) 58  
Zuweisungsoperator 86, 158  
    (=) 106  
    *überladen* 318, 327



Dieter Bär

## Schrödinger programmiert C++ – Das etwas andere Fachbuch

696 Seiten, broschiert, in Farbe, 2. Auflage 2015  
44,90 Euro, ISBN 978-3-8362-3824-3

 [www.rheinwerk-verlag.de/3892](http://www.rheinwerk-verlag.de/3892)

**Dieter Bär** kennt Schrödinger wohl am besten: War ein Bier mit ihm trinken und ist zu Recht stolz, dass sein Kumpel in die Riege der echten Entwickler aufgestiegen ist. Bär kann nicht nur C++, auch von C und Perl kann er die Finger nicht lassen.

*Wir hoffen sehr, dass Ihnen diese Leseprobe gefallen hat. Sie dürfen sie gerne empfehlen und weitergeben, allerdings nur vollständig mit allen Seiten. Bitte beachten Sie, dass der Funktionsumfang dieser Leseprobe sowie ihre Darstellung von der E-Book-Fassung des vorgestellten Buches abweichen können. Diese Leseprobe ist in all ihren Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen beim Autor und beim Verlag.*

*Teilen Sie Ihre Leseerfahrung mit uns!*

