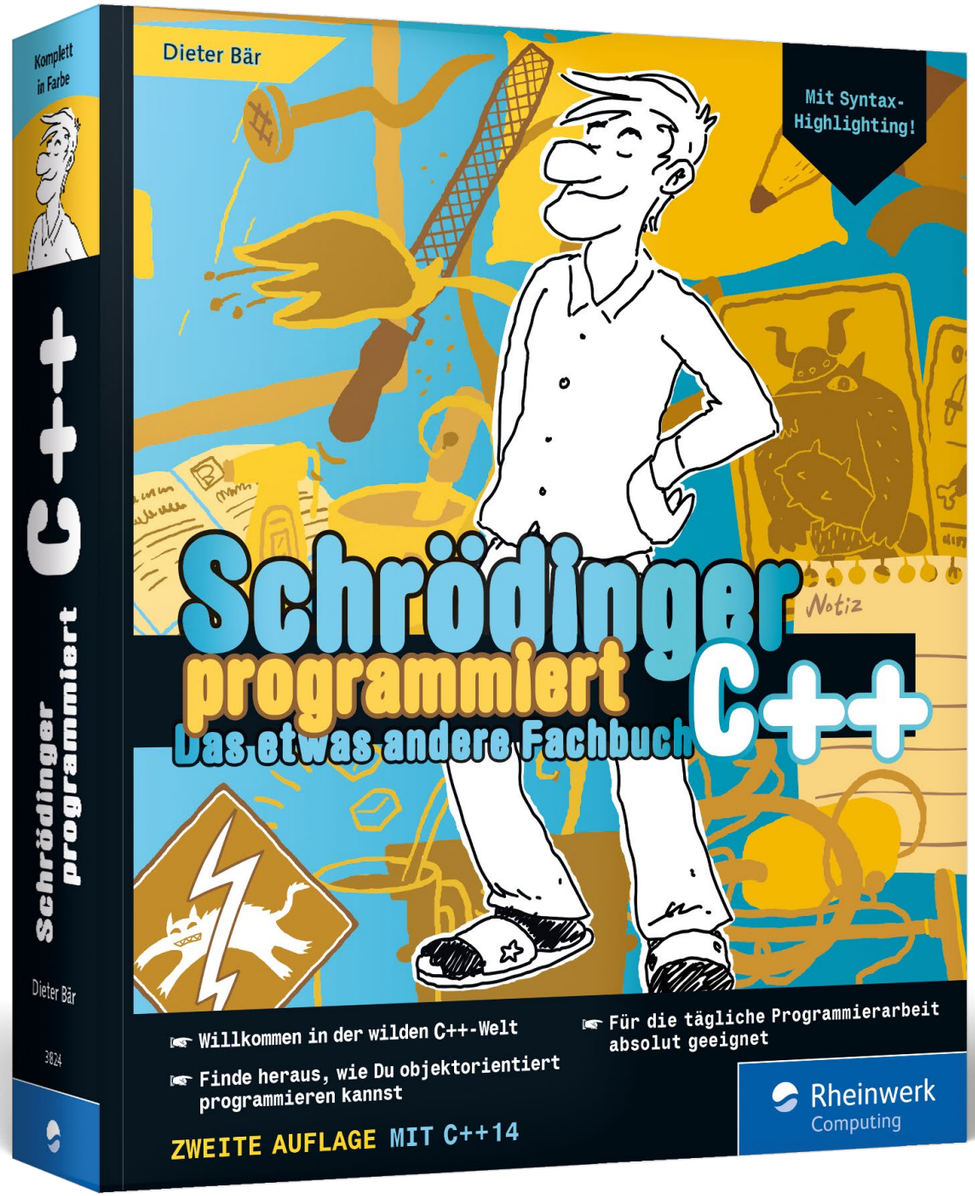
Schrödinger programmiert C++

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



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.

Der Autor

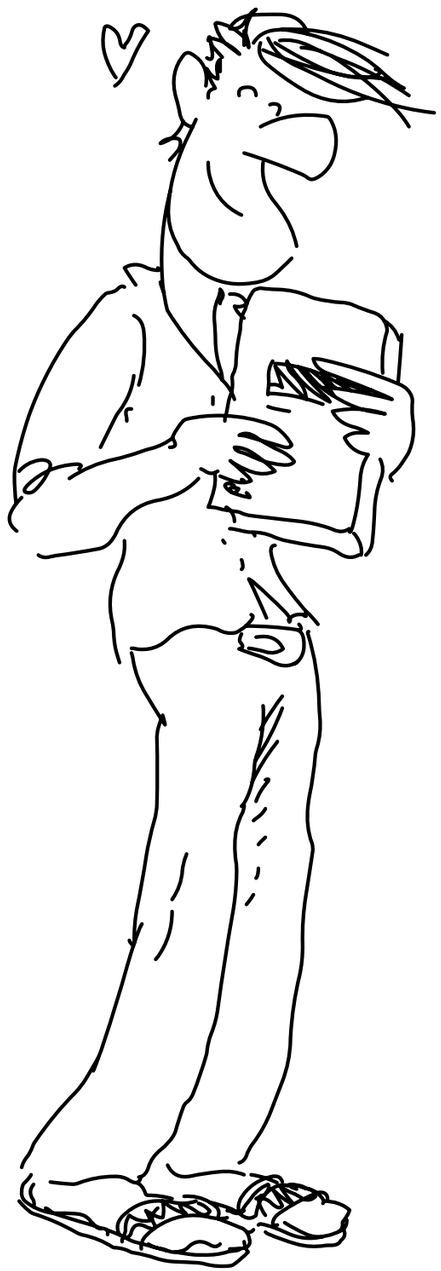
Leseprobe weiterempfehlen

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



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 Rumkugel 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?

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 ...

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

Der Verlag

Schrödingers Büro

Zettel/Notiz

Hintergrund/-info

Achtung/Vorsicht

Belohnung/Lösung

robots in disguise

Die nötige Theorie, viele Hinweise und Tipps

15:20

Begriffsdefinition

Ablage

Schrödingers Werkstatt

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

Schrödingers Wohnzimmer

Zettel

Belohnung/Lösung

Übung

Mit viel Kaffee, Übungen
und den verdienten Pausen

schwierige Aufgabe

einfache Aufgabe

Achtung/Vorsicht

—EINS—

Entwicklungs-
umgebungen
für C++

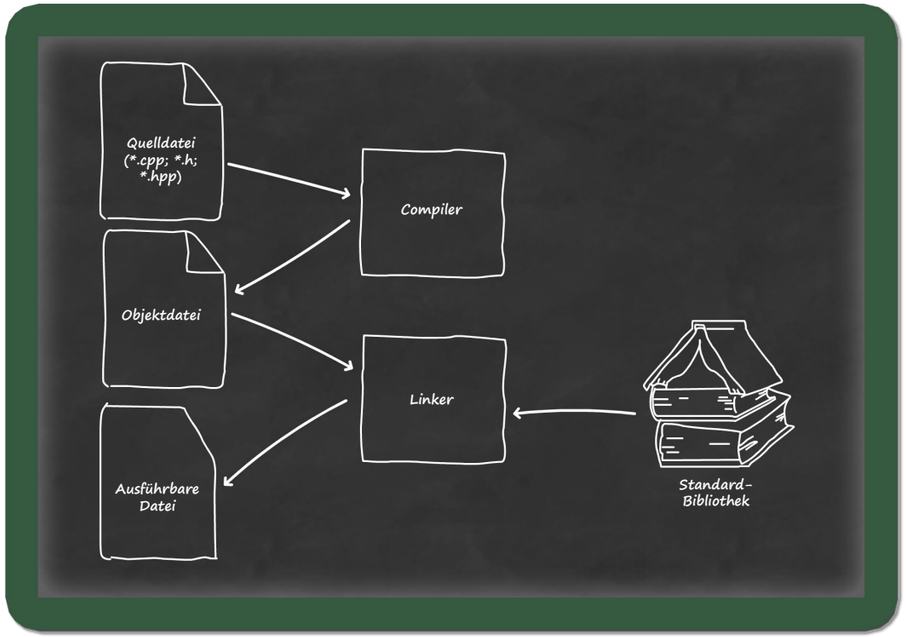
Wir richten uns ein …

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?

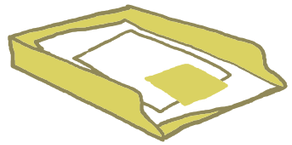
Okay, bevor du überhaupt anfangen kannst, deine Programme zu schreiben, brauchst du natürlich ein Werkzeug, um einen lesbaren Quellcode in einen nicht mehr lesbaren Maschinencode zu übersetzen. Für solche Zwecke benötigst du in C++ einen **Compiler**. Solange du den C++-Standard verwendest, ist dein 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

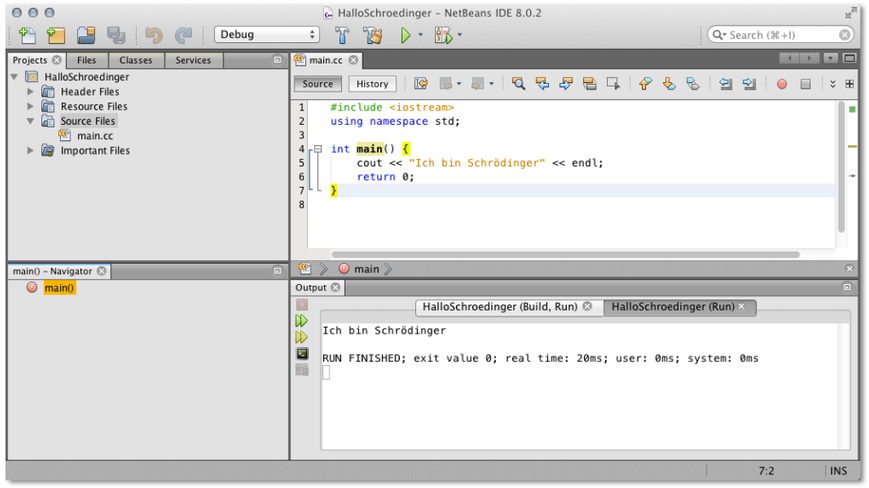
Um jetzt aus deinem Quellcode ein echtes Programm (eine ausführbare Datei) zu machen, brauchst du im Grunde nur einen **ASCII-Texteditor**, in den du deinen Quelltext eintippst, und dann eben einen **Compiler** (mit Linker) zum Übersetzen des Quelltextes, um daraus ein Programm zu machen.
Natürlich kannst 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-Komplettsachen ist halt, dass hierfür ein wenig mehr Einarbeitungszeit nötig wird.



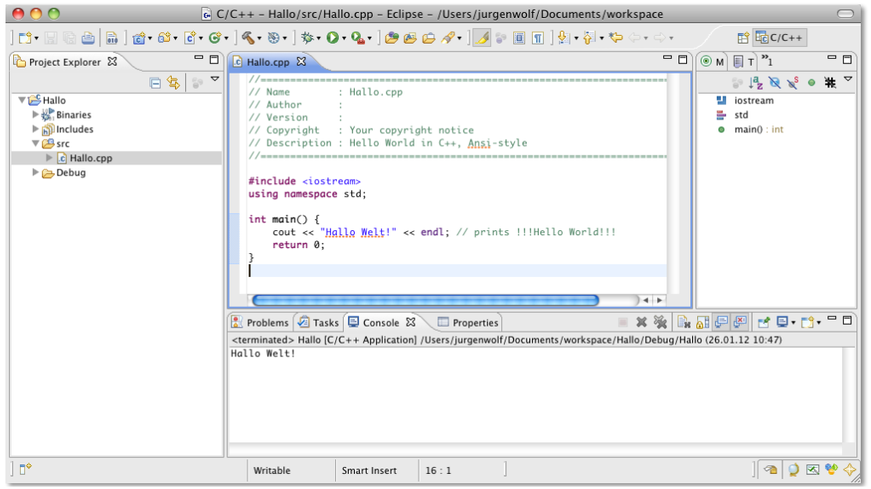
[Ablage]
Wie dem auch sei, ich bin hier nicht dafür da, für bestimmte Compilerhersteller zu werben, sondern ich will dir lediglich einen kleinen Überblick zu diesem Markt verschaffen. Letztendlich sind alle nur ein Mittel zum Zweck mit demselben Ziel, nämlich ein ausführbares Programm zu erzeugen.

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++.



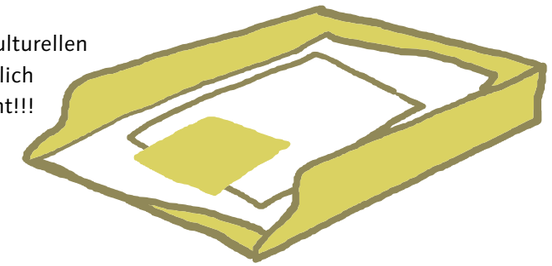
Die Entwicklungsumgebung NetBeans für C++



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.

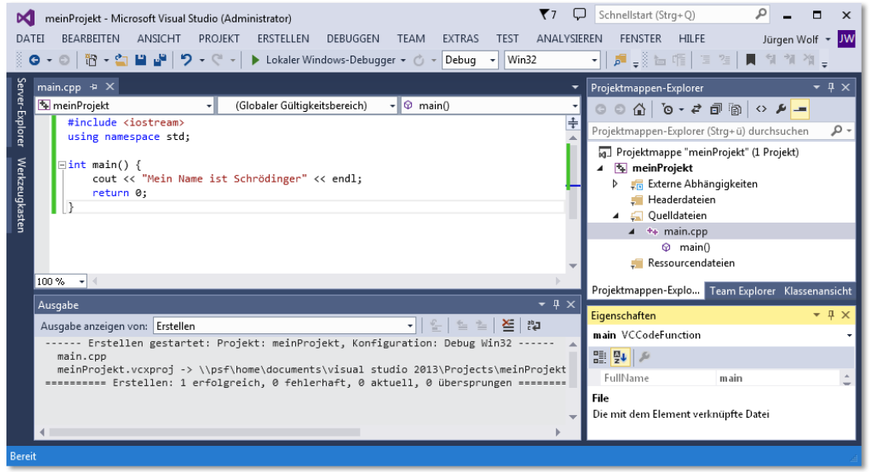
[Ablage]
Alle hier erwähnten multikulturellen Sachen sind umsonst erhältlich und kosten dich keinen Cent!!!



Mikroweiche Sachen

Dominierend auf den Windows-Rechnern dürften wohl die hauseigenen Produkte sein. Microsoft bietet hierbei unter der Hausmarke **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

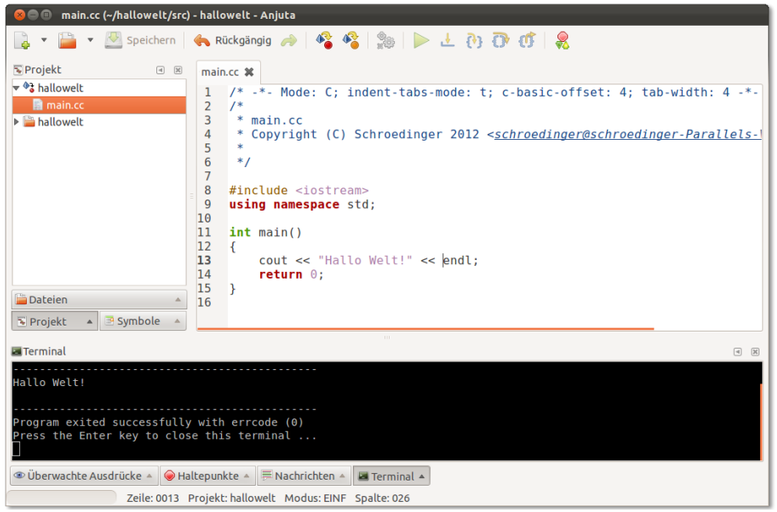


Auch sehr beliebt ist der Kommandozeilen-Compiler **MinGW-Compiler**, der eine Portierung des GCC-Compilers auf Windows ist. Darauf aufbauend werden viele andere Entwicklungsumgebungen verwendet. Früher war auch Borlands C++Builder sehr beliebt, der allerdings jetzt nur noch **C++Builder** heißt, weil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

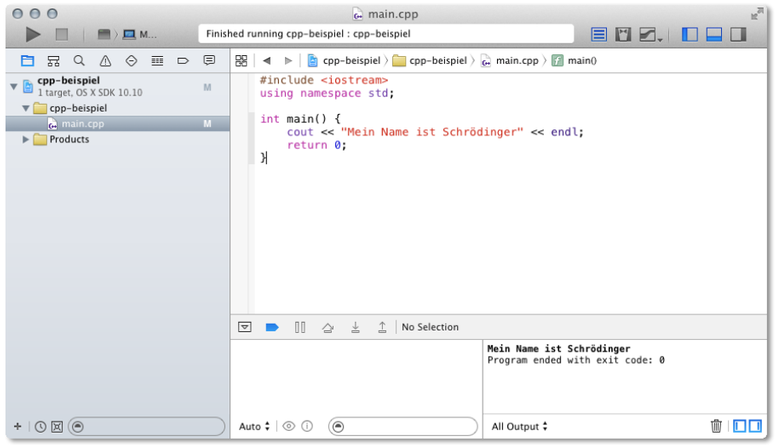
[Hintergrundinfo]
GCC, gcc und g++! Alles dasselbe? Nein, nicht ganz. GCC ist die GNU Compiler Collection (also eine Sammlung von Compilern). gcc (kleingeschrieben) ist der C-Compiler der Sammlung, und g++ ist (ja, richtig) der C++-Compiler von GCC!



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.

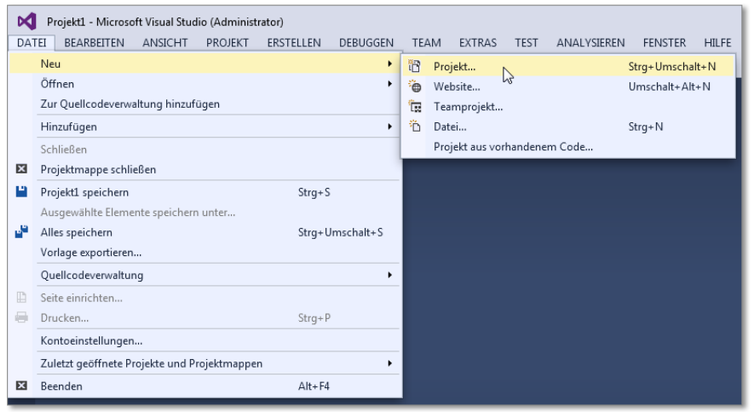
[Notiz]
Du musst nicht zwangsläufig wie Schrödinger lauter neue Rechner anschaffen, um deinen Code oder deine Beispiele eventuell auf anderen Betriebssystemen oder Compilern zu testen. Hierzu reicht häufig auch eine **Virtualisierung** aus. So kannst du bspw. auf dem Mac-Rechner über eine Virtualisierungssoftware 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.

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

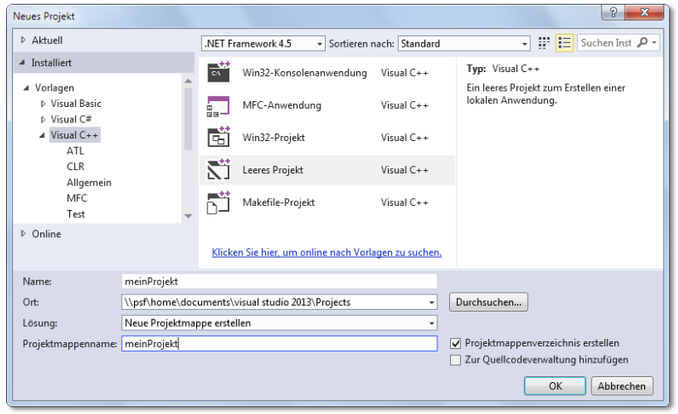
selbst ein wenig mit deiner Entwicklungsumgebung befassen musst. In der Abbildung wurde zwar Microsoft Visual Studio Community dafür verwendet, aber der Vorgang ist bei den meisten anderen Entwicklungsumgebungen auch sehr ähnlich aufgebaut. Nur dass hier und da der Befehl etwas anders lautet und sich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

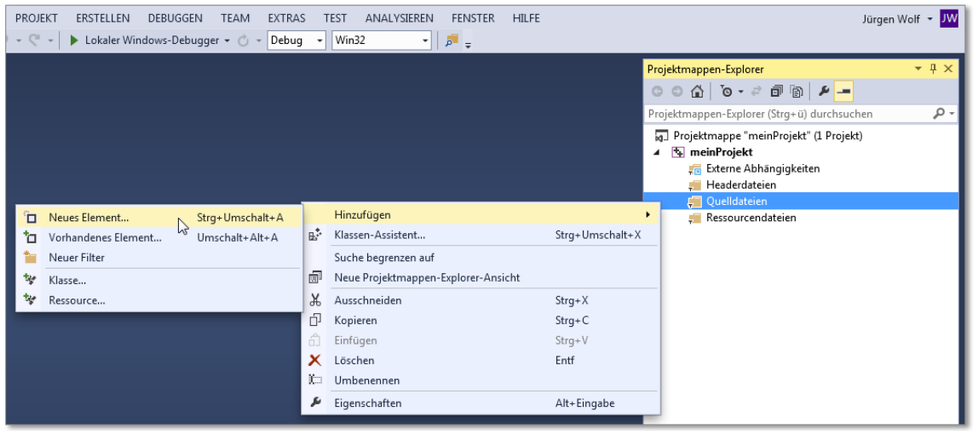
2. Als Nächstes erscheint meistens ein Dialog, der dir dabei hilft, zu entscheiden, was für eine Art von Projekt du erstellen willst, wie du dein Projekt nennen und wo du es abspeichern willst. Hier ist es oft gut, ein leeres Projekt oder eine C++-Konsolenanwendung zu erstellen.



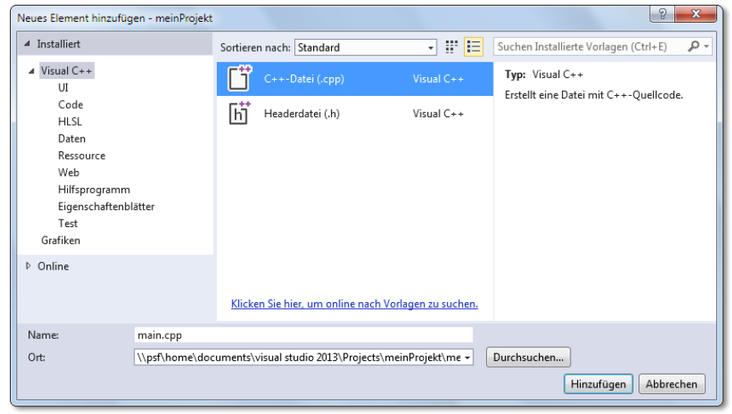
Gewöhnlich hilft dir jetzt ein Wizard weiter, die Art deines Projektes auszuwählen.

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.

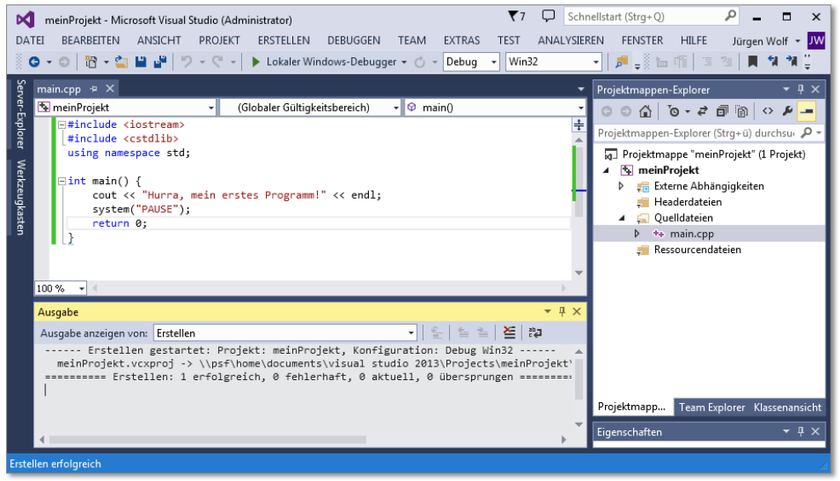


4. Auch hier hilft dir häufig ein weiterer Dialog bei der Wahl, was für eine Art von Datei du zum Projekt hinzufügen willst. Hier ist es eine Quelldatei mit der Endung **`*.cpp`**. Aber bei späteren Projekten wirst du auch des Öfteren mal eine Headerdatei mit der Endung **`*.h`** hinzufügen.



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.

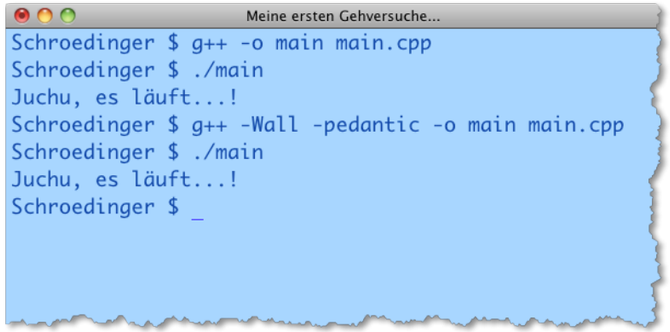
[Fehler]
Wenn du beim Einstieg hier verzweifelst und wirklich nicht klarkommst, kannst du ja eine E-Mail an Schrödinger senden, der da ziemlich gut durchblickt:

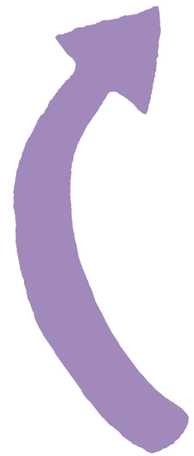
schroedinger@dieter-baer.de

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 hangelst. 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

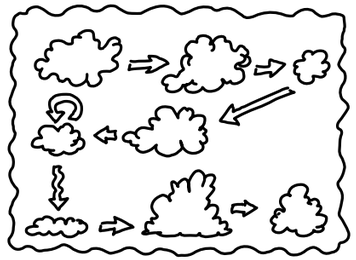
So, an der Auswahl von Compilern bzw. Entwicklungsumgebungen mangelt es ja nun wirklich nicht. Daher will ich dir hier noch ein paar Compiler bzw. Entwicklungsumgebungen auflisten, damit du einen Überblick hast. Du kannst diese gerne testen und selber sehen, was dir persönlich am meisten zusagt.

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

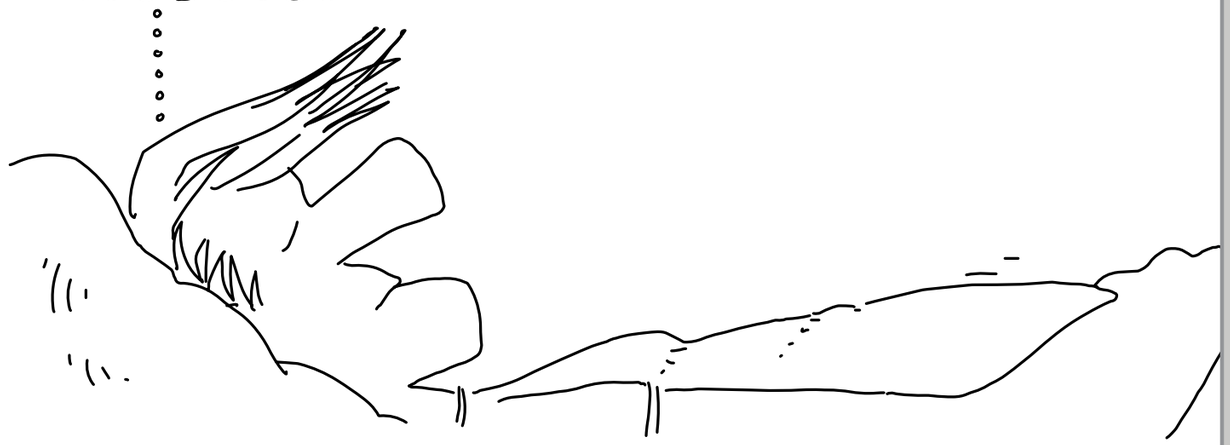
Übersicht der Compiler bzw. Entwicklungsumgebungen für C++

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 Binärcode, 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 alle 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:

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

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;
}
```

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?

Immer langsam, die Welt wurde auch nicht an einem Tag erschaffen (theologisch betrachtet in sieben – oder waren es sechs??? –, aber wissenschaftlich ...).

[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:

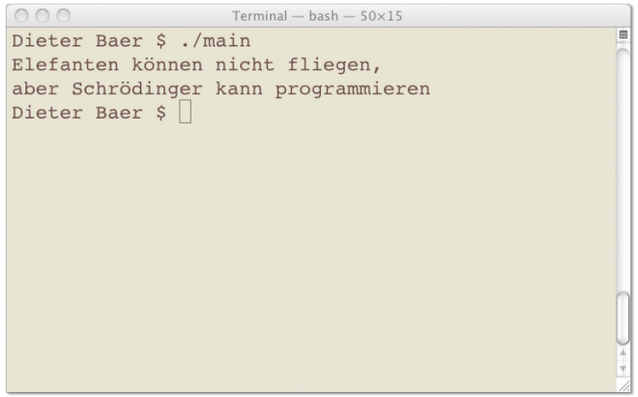
```
#include <iostream>

using namespace std;


int main()
{
 cout << "Elefanten können nicht fliegen,\n";
 cout << "aber Schrödinger kann programmieren\n";
  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.

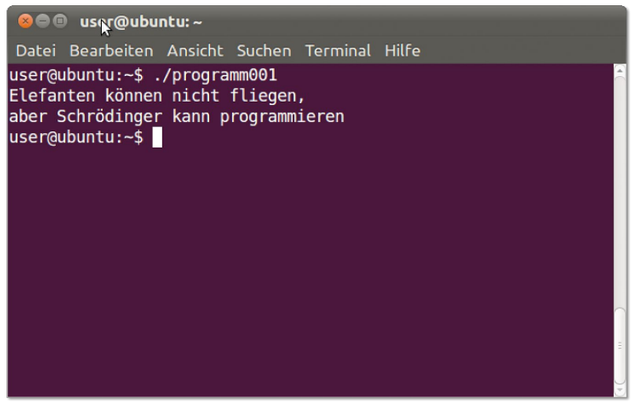
*2 Damit stellst du für dein Programm den **Namensraum `std`** zur Verfügung. **Das ist praktisch**, weil du ohne diese Zeile Befehle aus diesem Namensraum erst mit **`std::cout`** oder **`std::endl`** qualifizieren müsstest.

*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 ...



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



Heureka, mein Programm auf meinem Ubuntu-Linux

Und zu guter Letzt das ausführende Programm unter MS Windows

[Achtung] Wenn falsche Umlaute in der MS Windows-Boxangezeigt 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.

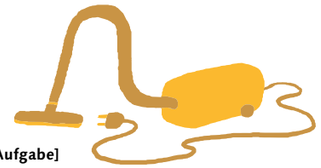
Einige Entwicklungsumgebungen kümmern sich selbst darum, bei anderen musst **du dich darum kümmern**. Im Grunde ist es ja kein Fehler, wenn das Programm wieder verschwindet. Letztendlich wird das Programm ja von Anfang bis Ende ausgeführt. Leider findet die Ausgabe hier ja auf die MS-DOS-Eingabeaufforderung 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!

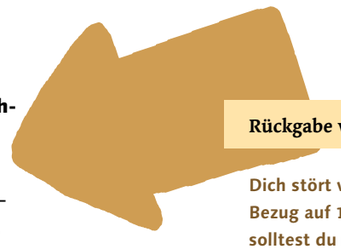
Wenn das weiterhin so einfach ist, könnte ich mich doch bei Blizzard Entertainment für die nächste Auflage von WoW bewerben. Das wäre doch eine Karriere, vom Entwickler für Software von Schuhkartons zum Chef-Entwickler eines Top-Software-Konzerns...



[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:

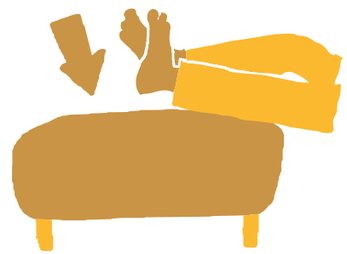
- Soll sich dein Kommentar nur über eine Zeile erstrecken, reicht die Zeichenfolge **`//`**. Alles, was sich jetzt dahinter befindet, wird vom Compiler ignoriert.

```
// 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.

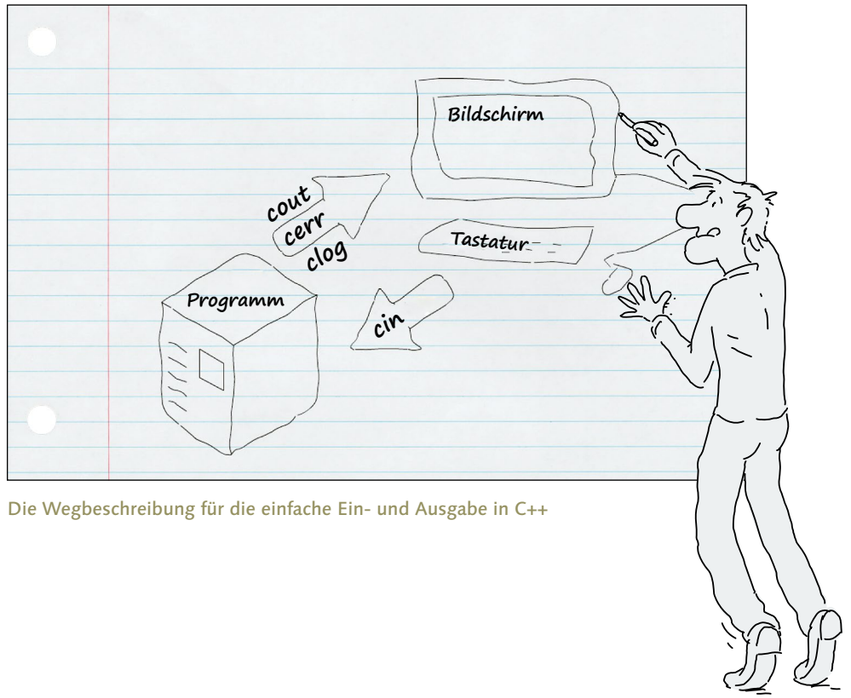
[Begriffsdefinition]

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

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
```

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

*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.

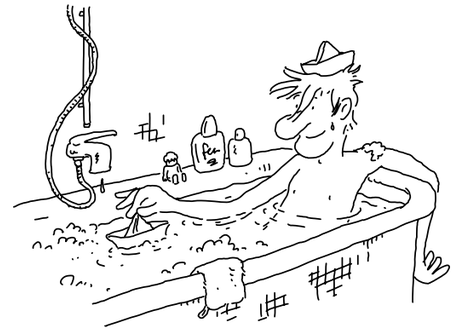
Badewanne schon voll?

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.

```
int gibmirfuenf; *1
cout << "Gib ihm fünf: ";
cin >> gibmirfuenf; *2
```

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

*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;

}
```

*1 Die Aufforderung für eine Eingabe.

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

*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 hüpft für uns in die nächste Zeile.

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.

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

```
#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;

}
```

*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.

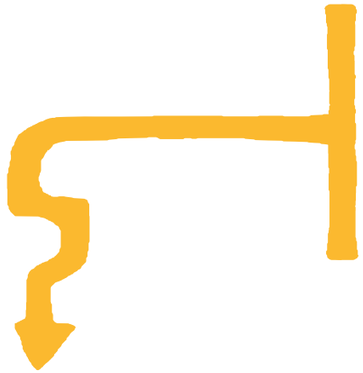
*2 Die wohl **sparsamste Lösung**, weil nur ein einzelnes Zeichen verwendet wird.

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

*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.

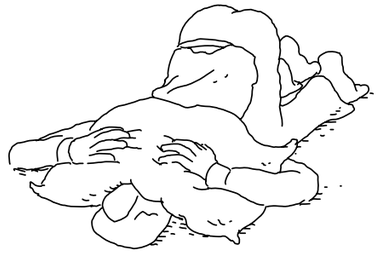


[Einfache Aufgabe]
Bevor du dich also auf **neue Quests** in der Welt der Kriegskunst stürzen kannst, möchte ich noch sehen, ob du das Thema verstanden hast. Du solltest jetzt in der Lage sein, im folgenden Listing ohne Lupe und Rechner, die Fehler mit dem **bloßen Auge** zu erkennen.

```
#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 hereingebeten 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;** hereingebeten 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—

Die C++ Basistypen

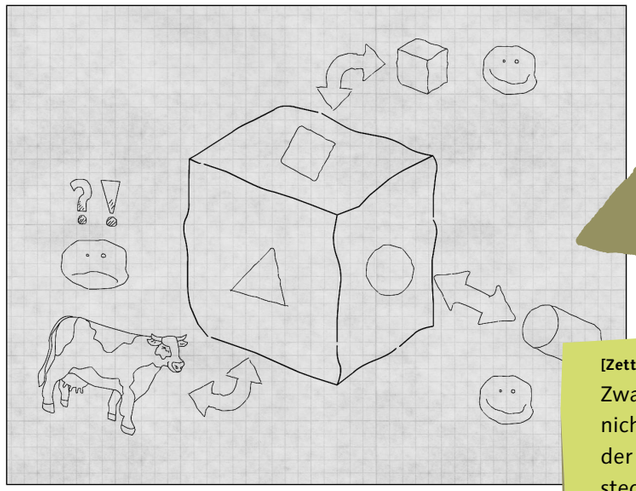
Verschiedene Typen für einen bestimmten Zweck

„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 ...



[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.

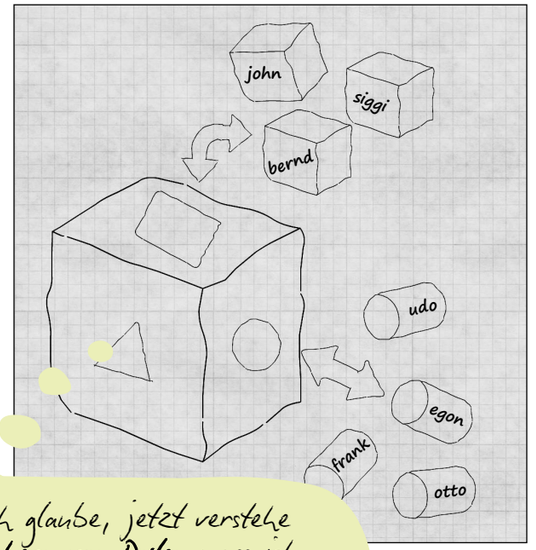
Ähnlich, wie du bei WoW als Priester heilen kannst und als Krieger eher nicht, legst du mit einem **Typ in C++** auch gleich fest, was du damit alles anstellen kannst. Schau dir einmal folgende Zeile an:

```
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.

Mein Name ist Schrödinger

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



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).

[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!**

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

Fundamentale und faule Typen

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).

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.

[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.

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.

Ganzer Kerl dank ...

>> Ein Blick in die Glaskugel

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.

[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.

[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
```

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 grundlegende 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 **`nena`** bekommt den Wert 99 **zugewiesen**.

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

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

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

```
int dieter = 36; *1
short kleinerDieter, nena = 99; *2
long grosserDieter; *3
cin >> grosserDieter; *3

int helmut = dieter; *4
long schroedinger; *5
schroedinger = 1234; *5
```

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 Gegenstücken.

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 Werteteil 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:

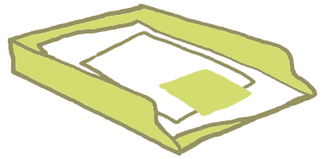
```
unsigned int positiverTyp; *1
int neutralerTyp01; *2
signed int neutralterTyp02; *3
```

*1 Die Variable dieses Typs kann wegen des Schlüsselworts **`unsigned`** **keine** negativen Werte speichern.

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

*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.

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`**). Kombinierst 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 **U**s und **L**s 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.

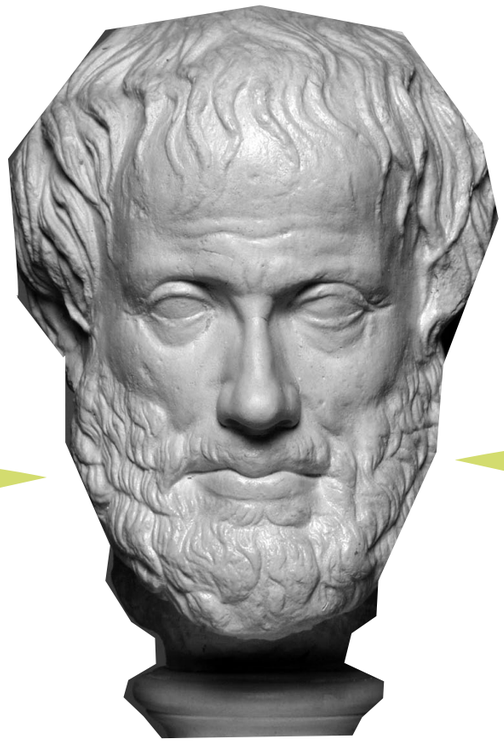
```
int dieter=1974, eva=1980; *1
bool treu = dieter==eva; *1
bool ichLiebeDich = 1234; *2
int liebe = ichLiebeDich; *3
int wahreLiebe = 0;
bool iLoveYouBaby = wahreLiebe; *4
```

*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.

Nicht darum nämlich, weil unsere MEINUNG, du seiest 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

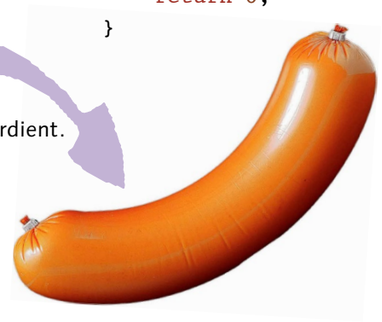
```
#include <iostream>
using namespace std;

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;
}
```

*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.

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

[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:

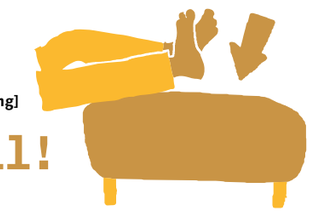
- Du brauchst einen **eindeutigen**, **gültigen Namen** (Bezeichner), der aus beliebigen Buchstaben, Ziffern und dem Unterstrichzeichen bestehen darf. Dabei wird auch zwischen **Groß- und Kleinschreibung** unterschieden!
- Wie bei Legobausteinen gibt es in C++ zunächst nur **fundamentale Typen** (Basistypen). 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:

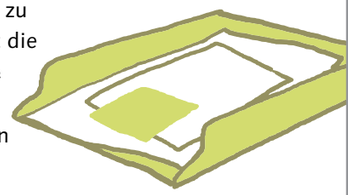
```
123.456  .33  0.33  2.  2.0 2.2e10
2.22e-14
```

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 Cpp11newstandard{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;
}
```

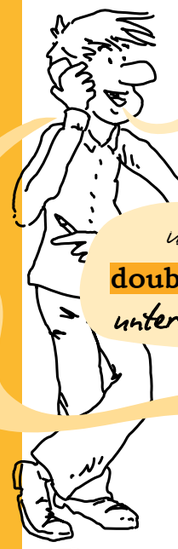
*1 Bei diesem sinnfreien Beispiel kannst du genauso gut den Typ `float` oder `long double` statt `double` verwenden. Ich habe mich halt gleich für **Daddys Liebling** entschieden.

*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!**

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 ...

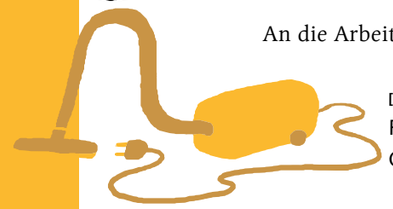


In C++ wird nicht das Komma, sondern das Punktzeichen verwendet, und es gibt die drei Typen `float`, `double` und `long double` mit einer unterschiedlichen Genauigkeit.

Ach, Schrödinger, schonmal was von Gleitkommazahlen gehört? `double` ist ja gut und schön, aber über Gleitkommazahlen musst du dich schlau 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;
```

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

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

*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.

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

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

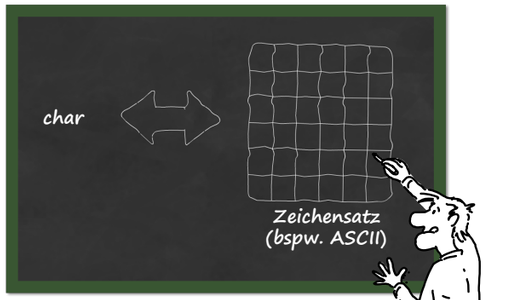
[Belohnung/Lösung]
Jetzt ist aber eine dicke Belohnung drin: die neue Blu-ray „Angriff der achtbeinigen Monster"!

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.

[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.

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.



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;
```

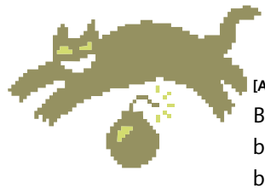
Dann ist es also egal, ob ich das Ding mit den Hochkommas oder direkt den dezimalen Wert verwende?

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 Rechner 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 **üääößÜÖÄ** oder die anderen europäischen landestypischen Zeichen.

Eine Gruppe von Leuten (die ISO) haben sich darum gekümmert. Sie haben den ASCII-Zeichensatz auf **8 Bits** aufgebohrt 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'`**.

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` `\UXXXXXXXX`	für Zeichen eines größeren Zeichensatzes (bspw. Unicode)

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"

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.

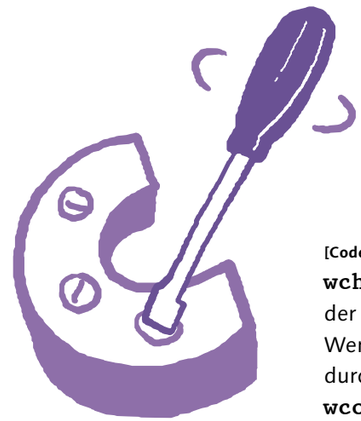
*2 Mithilfe des Ausdrucks `int(ch)` kannst du den **ganzzahligen Wert** der Zeichen `'A'` und `'Z'` auf dem Bildschirm ausgeben.

*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.



Hrmpf! Ich habe das Beispiel eben mit `wchar_t` getestet. Jetzt werden nur noch Ganzzahlen statt der Zeichen ausgegeben!?

[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`.

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

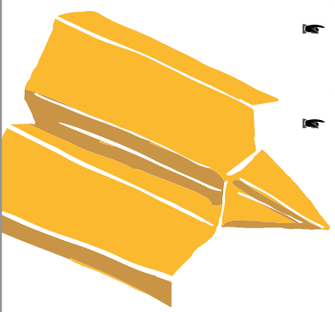
Nun ja, Schrödinger! Das Thema mit den Zeichensätzen mag dir am Anfang wie der Turmbau zu Babel vorkommen. Aber mittlerweile sieht es gar nicht mehr so schlimm aus. Spätestens seit dem C++11-Standard ist es mit den neuen Typen `char16_t` und `char32_t` und der Unicode-Welt erheblich einfacher geworden, 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. Umlauten) 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;
```

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

*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.

*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 Backslashs (\) wird hier ein normaler Slash (/) verwendet.

[Belohnung/Lösung]
So, jetzt kannst du dich ins Vergnügen stürzen.

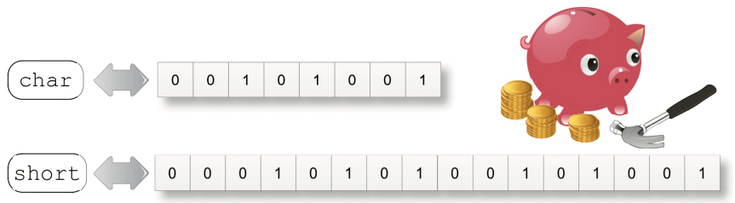
Auf die Größe kommt es an ...

Wenn du stundenlang WoW spielst oder wenn du dich in der realen Wert 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 hoch 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 einführten 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
```

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

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

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 **<cfloat>** 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
`bool`	1 Byte	0 oder 1 bzw. `false` oder `true`
`char`	1 Byte	–128 bis +127 bzw. 0 bis 255
`signed char`	1 Byte	–128 bis +127
`unsigned char`	1 Byte	0 bis 255
`wchar_t`	2 oder 4 Bytes	abhängig von der Implementierung
`short`	2 Bytes	–32.768 bis +32.767
`unsigned short`	2 Bytes	0 bis 65.535
`int`	4 Bytes	–2.147.483.648 bis +2.147.483.647
`unsigned int`	4 Bytes	0 bis 4.294.967.295
`long`	4 oder 8 Bytes	wie `int` oder –9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
`unsigned long`	4 oder 8 Bytes	wie `unsigned int` 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

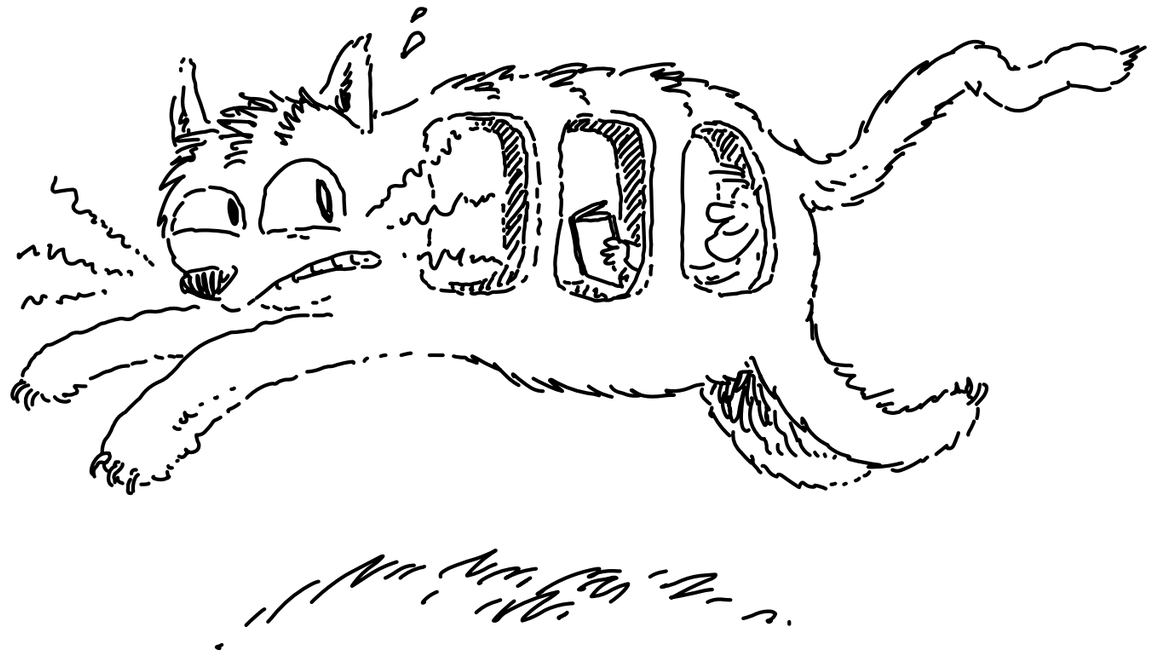
Jetzt fehlt dir nur noch der Überblick über die Gleitkommatypen:

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* `9223372036854775807` *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;
```

INHALTSVERZEICHNIS

Kapitel 5: Eigene Entscheidungen treffen oder das Ganze nochmal bitte

Kontrollstrukturen in C++

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

Arrays, Strings, Vektoren, Strukturen und Zeiger

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

Funktionen

Kapitel 8: Wie aus Chaos Ordnung entsteht

Schlüsselwörter für Typen, Namensbereiche und die Präprozessor-Direktiven

Kapitel 9: Von Hexenmeistern, Todesrittern und Datenkapseln

Klassen

Kapitel 10: Kino + WoW + Programmieren = viel Spaß

Überladen von Operatoren

Kapitel 11: Schrödinger macht sein Testament

Abgeleitete Klassen

Kapitel 12: Ausstechformen für die Plätzchen

Templates

Kapitel 13: Der Schleudersitz für den Notfall

Ausnahmebehandlung

Kapitel 14: Unterwäsche, 100 % Baumwolle, Doppelripp

Die Standardklasse string

Kapitel 15: Ströme ohne Isolierkabel verwenden

Der Umgang mit Streams und Dateien

Kapitel 16: Ausstechformen für Faule

Einführung in die Standard Template Library (STL)

Kapitel 17: Schöne neue Welt C++11

C++11 – der neue Standard

Kapitel 18: C++ 14 – der Neue!

C++14 – der allerneueste Standard

INDEX

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

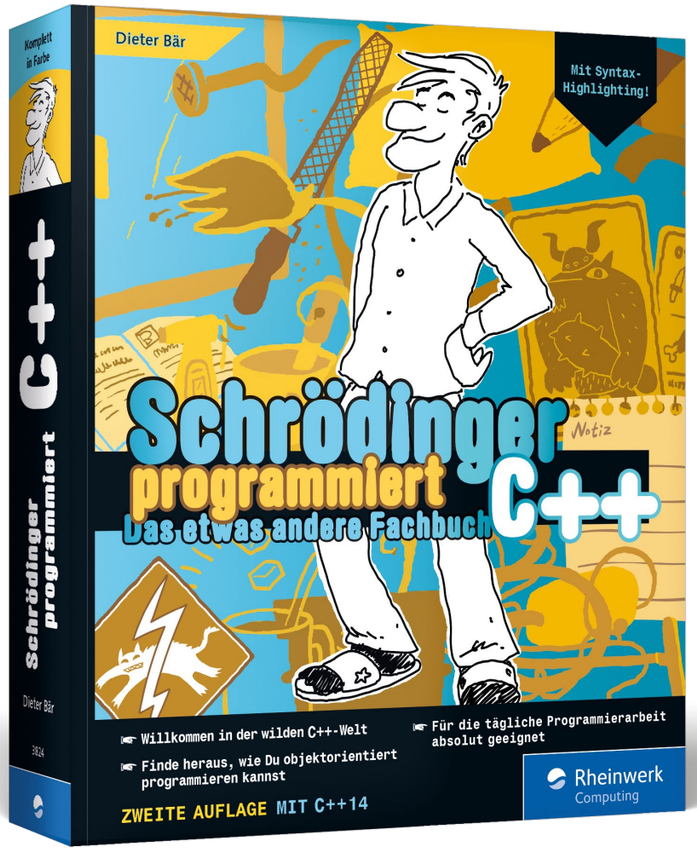
O

P

V

W

Z



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

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.