C++ Kurs für totale Anfänger

Herzlich Willkommen bei meinem ersten C++ Tutorial! Hier will ich Anfängern den Einstieg in die wohl meistgenutzte Programmiersprache der modernen Softwareindustrie ermöglichen. Ich hoffe natürlich, dass ich auf die einzelnen Themen ausreichend genau eingehen werden kann. Gleich vorneweg: Dieser Kurs ist wie eigentlich jedes Tutorial eine recht komplexe Sache, zumal es sich um C++ handelt. Ich bin noch nicht fertig, ihn zu schreiben - und bis er perfekt ist, wird noch ein Weilchen vergehen! Damit du keinen Schiffbruch während dieser Phase erleidest, kannst du auf meiner Homepage rumwühlen - **Hier gehts lang**.

Ich will dich jetzt nicht lange mit historischem Zeug aufhalten, aber hier vielleicht kurz was zur Sprache (der Allgemeinbildung wegen): C++ ist eine Weiterentwicklung der Programmiersprache C; diese wurde '72 bei der Firma Bell Laboratories von Dennis Ritchie entwickelt. Bjarne Stroustrup aus der selben Firma entwickelte daraus die C++ Programmiersprache, die C um einige komfortable Funktionen, aber noch wichtiger – um die Objektorientierung – erweiterte. '98 wurde dann ein Standard für C++ verabschiedet - ANSI/ISO C++.

Gliederung

Dieses Tutorial wird in 2 Versionen vorliegen – als DOC/PDF und als HTML Dateien. Gründe dafür:

DOC oder **PDF**, weil dieses Format besser zum Ausdrucken ist (HTML interpretieren die Browser nicht alle gleich). So kannst du nachts noch ein bisschen lesen, wenn du nicht schlafen kannst, und schon mal eine Seite vorausgucken. Außerdem kannst du so viel besser lernen, als wenn du immer zwischen dem Programm (eigentlich Compiler - dazu später) und einem Kapitel hin- und herwechseln musst. Ein Vorschlag zum Ausdrucken und Binden findest du im Anhang.

HTML, weil du hier die Vorteile von Links nutzen kannst. Du kannst also durch Klicken auf den Link sofort ein Projekt öffnen, ohne ständig zwischen dem Compiler und deinem Explorer wechseln zu müssen. Denn das ist eine Sache, die auf die Dauer kräftig an die Nerven gehen kann.

In jedem Kapitel wird ein einzelnes Thema behandelt. Am Ende eines jeden Kapitels wird es eine kleine Zusammenfassung und ein paar Aufgaben geben. Das Tutorial ist so konzipiert, dass es im vorgegebenen Ablauf durchgelesen werden sollte. Außerdem findest du am Ende jedes Kapitels Links zu Tutorials, die ähnliche Kapitel haben.

So, nun will ich dir mal was zum Programmierer erzählen:

Du hast dich sicherlich schon mal gefragt, wie moderne Softwareprogramme entstehen. Jeder kennt Spiele und Programme, die einfach nur genial sind (Diablo 1&2, Counterstrike ...). Wie macht man also solche Programme und Spiele???

Jedes Programm muss, bevor es funktionieren kann, natürlich erst mal programmiert werden. Dazu gibt es halt die Programmiersprachen (neben C++ und C auch noch Pascal und Delphi und Basic und viele weitere...), die, mittels geeigneter Programme, den vom Programmierer geschriebenen Code in ein vom Computer ausführbares Programm umwandelt. Die kann man mehr oder weniger leicht und schnell lernen, je nach dem, welche Informationsquellen man hat. Doch die professionelle Programmierung ist garantiert nicht einfach: moderne Softwareprodukte haben schnell mal MEHRERE 10000 Zeilen in ihren Quellcodedateien. Aber bis dahin ist noch ein bisschen Zeit!

Sehr gut geeignet, um Vorurteile, Illusionen und Erwartungen zu berichtigen, ist meiner Meinung nach der Aufsatz von Sebastian Porstendorfer: **Was einen Programmierer ausmacht**:

Ein Dokument der Weisheit:

Freundlicherweise vom Schreiber Sebastian Porstendorfer zur Verfügung gestellt (Danke)

Hallo und herzlich Willkommen zu meinem kleinen Essay

Newbie's Guide to Programming

Was ich hier vermitteln möchte, ist nicht irgendeine Programmiersprache oder ein Programmierstil oder ähnliches. Vielmehr möchte ich mich mit der Frage beschäftigen:

Was macht einen Programmierer aus?

Leider ist es gerade in der von mir angepeilten Zielgruppe oftmals so, dass sich einige falsche Vorstellungen über das Programmieren herausbilden. An dieser Stelle möchte ich dann auch gleich eine Warnung aussprechen: Für manchen wird die Lektüre dieses Textes nicht gerade motivierend erscheinen, geschweige denn wird es ihn erfreuen zu lesen, dass er eigentlich lieber gleich die Programmierung sein lassen sollte. Dieser Text ist keine Do-it-yourself-Anleitung, die, nachdem man sie durchgearbeitet hat, sofort ungeheures Wissen in eure Köpfe presst. Dieser Text mag unbequem sein, aber das ist die Arbeit als Programmierer oftmals auch. Fall sich also jemand von diesem Text enttäuscht zeigen sollte, bitte ich darum, keine Hassmails auf mich loszujagen, denn ich berichte nur über die Realität und will etwas Licht ins Dunkel des Programmierdjungels bringen.

Dass ich dabei einigen ihre Illusionen nehme, ist ein durchaus beabsichtigter Nebeneffekt, der aber von ihnen nicht negativ gesehen werden muss: Nehmen sie es als Anreiz, ihre Motivation, Programmieren zu lernen nochmals zu überdenken.

Um zu den angesprochenen falschen Vorstellungen zurückzukommen und zugleich einen Einstieg in meine Ausführungen zu finden (das etwas hohe Sprachniveau in diesem Essay ist btw. Durchaus beabsichtigt, da es für diejenigen, die über die nötige Intelligenz und den nötigen Sachverstand vermögen, ganz sicher kein Problem darstellen mag, einen in höherem Schriftdeutsch geschriebenen Essay zu verstehen und es gleichzeitig auch eine Art Vorbereitung auf das Niveau "richtiger" Programmierbücher abseits der "in 21 Tagen" und "xxx für Dummies" Reihen darstellen soll), möchte ich nun mit einem gerade unter Neueinsteigern verbreitetem Vorurteil aufräumen: Programmierer wird man nicht, indem man ein Buch, ein Tutorial (wobei diese eher 2. Wahl darstellen) oder sonst etwas liest, sondern indem man sich mit der Sprache auseinandersetzt.

Oftmals werden Programmiersprachen mit realen Sprachen verglichen, so dass ich diesen Vergleich hier nun einmal aufgreifen möchte. Wie bei einer realen Sprache muss man verschiedene Dinge lernen und beherrschen. So gibt es auch in einer Programmiersprache "Vokabeln", dass sind die Anweisungen und Komponenten der Sprache, als auch "Grammatik", also Syntax der Programmiersprache und das Wissen um die Bedeutung des Kontextes. Wie bei einer realen Sprache auch ist es unsinnig, nur die Vokabeln oder nur die Grammatik zu lernen, da das Eine ohne das Andere keinen Sinn macht. Was nützt es mir, jede französische Vokabel zu kennen, ohne einen einfachen Satz bauen zu können?

Nichts.

Genau so ist es bei einer Programmiersprache auch. Übrigens ist auch die Lernmethode prinzipiell nicht anders als bei einer realen Sprache auch. Auch eine Programmiersprache kann ich durch Anwendung dieser viel schneller verstehen lernen als durch pures Auswendiglernen. Nur durch Auswendiglernen kann man noch lange nicht programmieren.

Was reale Sprachen von Programmiersprachen unterscheidet, ist, dass kein Dialog stattfindet. Das heißt, sie sagen dem Computer zwar was er tun soll, das heißt aber noch lange nicht, dass er ihnen auch etwas anderes mitteilen will, als das, was sie ihm sagen, was er ihnen mitteilen soll. In der Tat ist alles, was er ihnen mitteilt, von ihnen bestimmt, da sie ihn per Anweisung auffordern, irgendetwas auf dem Bildschirm oder sonst wo auszugeben. Von daher müssen sie ein Programm nur verstehen können, indem sie es lesen, und wissen, was es bewirkt, aber sie müssen es nicht wie der Computer übersetzen, weil sie dem Computer ja praktisch beim Programmieren ihre Sprache aufzwingen, und er selbst dann die Übersetzung vornimmt.

Wie sie vielleicht wissen, erfordert es eine Menge an Aufwand, um eine Fremdsprache zu lernen. Sie müssen regelmäßig ihre Vokabeln lernen, sie müssen sich am Anfang genau über die Bedeutung von diesem oder jenem Gedanken machen, aber sobald sie einen gewissen Grad an Erfahrung erreicht haben, relativiert sich der Aufwand und sie müssen sich nicht mehr mit den grundlegenden Mechanismen auseinandersetzen, weil sie sie gelernt haben. Und nun raten Sie mal, wie man eine Programmiersprache am effektivsten lernt?

Wenn Sie noch keinen blassen Schimmer von der Programmierung haben und ihnen die meisten Vokabeln unbekannt sind, werden sie viel Zeit damit zubringen, indem sie versuchen zu verstehen, wieso etwas so und so funktioniert und nicht anders. Sie werden es mitunter nicht immer als Vergnügen ansehen, aber sobald sie einen gewissen Kenntnisstand erreicht haben, können sie anhand der Dinge, die sie bereits kennen, auf die Funktion der unbekannten Dinge schließen und ihre Anstrengung so minimieren. Spätestens zu diesem Zeitpunkt wird ihnen die Tätigkeit als Programmierer Spaß machen. Sie kennen viele Dinge und es entsteht automatisch eine magische Neugier auf die Dinge, die sie noch nicht kennen in ihnen. Auch komplexe Zusammenhänge erscheinen ihnen immer klarer. Am Anfang hingegen steht erst mal langweiliges Erlernen der Grundlagen. Das ist überall so, und wer hier nicht die Zähne zusammenbeißt und durchkommt, der mag vielleicht frustriert sein, aber, und das versichere ich ihnen, bei weitem nicht so frustriert als wenn er gleich zu Beginn ein 10000-Zeilen Programm vorgesetzt bekommt und einen Tippfehler oder ähnliches Suchen muss. So eine Fehlersuche ist das frustrierendste, was die Programmierung zu bieten hat, aber sie gehört dazu. Fehlerfreie Programme gibt es nicht. Auch wenn wir versuchen Fehler zu vermeiden, irgendwo machen wir doch welche. Das ist nicht schlimm, aber natürlich ist so ein "Showstopper" auch nicht gerade etwas, was für Heiterkeit sorgt.

Deshalb komme ich nun zum zweiten Punkt:

Programmieren ist keine Hexerei, sondern in gewissem Sinne eine Wissenschaft. Ein Programmierer sollte wissen, was er tut, und warum etwas so funktioniert, wie es funktioniert. Um das zu erreichen, ist Konzentration und Sorgfalt nötig. Das lässt sich schon dadurch erreichen, dass der Code lesbar gestaltet wird. Nicht ist schlimmer als unformatierter Code, der einfach straight untereinander geschrieben wurde, ohne ihn sinnvoll zu unterteilen oder ihn zu kommentieren und seine Funktion zu erklären.

Schon diese recht einfachen Maßnahmen helfen, so manchen Bug zu vermeiden, und ist er erstmal drin, so ist er in einem in Sinnabschnitte und formatierten Quelltext sicherlich leichter aufzuspüren als in einer endlosen, verwirrenden

Textwüste. Kommentare sind nicht nur für Feiglinge oder Dünnbrettbohrer, sondern dienen einfach einer kurzen Erklärung und Beschreibung des Programmteils, in dem sie stehen. Es mag einige Programmierer geben, die es vorziehen, sich x-fache Arbeit zu machen, als dass sie kommentieren oder formatieren. Sie als angehender Programmierer sollten darauf achten, dass der Code so gut lesbar und verständlich wie möglich ist.

Ein weiteres Manko ist, dass viele Newbies, die in dem nun geschilderten Fall eigentlich als "Lamer" bezeichnet werden, es sich sehr einfach machen möchten, indem sie auf vorgekaute Lösungen von anderen, engagierten Menschen hoffen, die ihnen hoffentlich gleich funktionierenden Code liefern, mitsamt einer Beschreibung wie er zu benutzen ist. In der Tat wird man mit so einem Verhalten auf wenig Gegenliebe stoßen, wie die vielen Flames in den einschlägigen Foren beweisen. Die Leute dort, die einem einen Lösungsansatz liefern, und davon ausgehen, dass der Hilfesuchende mit Hilfe dieses Ansatzes und vielleicht einer Beschreibung der Vorgehensweise in der Lage ist, daraus die Lösung für das Problem zu entwickeln kann, werden im Gegenteil oft genug noch als "arrogant" und "herablassend" bezeichnet, weil sie die Meinung vertreten, dass man Programmieren nur dadurch lernen kann, dass man es tut, und nicht dadurch, dass man einfach ohne Nachdenken vorgekauten Code übernimmt, ohne sich damit zu befassen. Ein derartiges Verhalten zeigt vor allem eines: Derjenige hat kein Interesse am Programmieren und an der Kunst, die das Programmieren darstellt, sondern will vielmehr nur wenig tun und dafür auch noch bewundert werden. Das dies der falsche Weg ist, dürfte aus meinen Ausführungen jawohl hervorgegangen sein.

Ich habe am Anfang des Artikels bereits gesagt, dass dies hier kein Patentrezept ist. Es gibt solche Patentrezepte nicht. Niemand lernt von heute auf morgen Programmieren, das ist ein Prozess der sich oftmals über einen langen Zeitraum erstreckt. Ja, im Prinzip lernen wir alle immer weiter, denn sobald wir aufhören zu lernen, hieße das ja, dass wir alles könnten. Das würde aber gleichzeitig bedeuten, dass man damit aufhören kann, da es keine neuen Herausforderungen gibt, denen man sich stellen kann, und letztlich stellt jedes noch so kleine Projekt eine solche Herausforderung dar. Die Herausforderung und der Willen zu Lernen sind der Hauptantrieb des Programmierers, ebenso wie der Wunsch nach der absoluten Kontrolle über die Maschine, und eben nicht das profane Geld, das nicht wirklich genauso glücklich macht wie das Erreichen eines Ziels dass man sich selbst gesteckt hat.

Ihr erstes Ziel sollte sein, eine Programmiersprache zu erlernen, und es spielt dabei prinzipiell keine Rolle, welche. Auch wenn es oftmals anders aussieht, wer wirklich etwas auf dem Kasten hat wird akzeptiert und angesehen und kann etwas bewegen, egal welche Sprache er benutzt oder welche Art von Software er schreibt. Allein das Können und der Lernwille zählen.

In diesem Sinne wünsche ich ihnen einen schönen Tag (oder eine gute Nacht) und möchte ihnen viel Glück wünschen, wenn sie immer noch den Wunsch haben, Programmieren können zu wollen. Wenn Sie es wirklich wollen, werden sie es auch schaffen, dann wird sie nichts aufhalten können.

Sebastian Porstendorfer

sp_sebisoft@gmx.de

PS : Bitte entschuldigen Sie die sicher vorhandenen Rechtschreibungs- und Grammatikfehler. Nobody is perfect.

Du wirst erfahren haben, dass du mehr tun musst, als nur dieses Tutorial durchzulesen und die Beispiele mal auszuprobieren. Wenn du lernen willst, gut zu programmieren, solltest du immer wieder kleine Programme programmieren, die das einsetzen, was du gerade gelernt hast. Nach und nach werden die Programme immer komplexer, und du wirst dir deinen eigenen Programmierstil bilden. Je mehr du übst, desto größere

Programme kannst du dir im Kopf mit deiner Logik aufbauen und dann (am Computer) testen, ob sie funktionieren (in der Tat wächst mit zunehmendem Können und Wissen die Fähigkeit, im Programme Kopf zu schreiben und auszuführen :).

Ich habe Programmieren mit einem Buch gelernt. Die meisten anderen Leser dieses Buches waren binnen weniger Wochen schon bei der Grafikprogrammierung mit DirectX - diese Leute haben C++ wohl nicht so richtig gelernt und geübt, was sich durch Forum-Einträge zeigte, die meist durch fehlendes Grundwissen verursacht wurden.

"Programmierer wird man nicht, indem man ein Buch, ein Tutorial (wobei diese eher 2. Wahl darstellen) oder sonst was liest, sondern indem man sich mit der Sprache auseinandersetzt." - beispielsweise wäre da die Beteiligung an Programmierforen.

Ich will Spieleprogrammierer werden, WAS MUSS ICH TUN???

Um überhaupt ein Programmierer moderner Software zu werden, musst du erst mal die Grundlagen kennen. C++ wurde in einer Zeit entwickelt, als es noch kein Windows 95, noch nicht einmal 3.11 gab. Es gab also keine Multitasking-Betriebssysteme wie heutzutage. Bis 1990 war DOS das Betriebssystem, mit dem die meisten Leute arbeiteten und spielten. Die Grundlagen von C++ beginnen demzufolge bei DOS.

Nachdem du diese DOS Grundlagen gelernt hast, kannst du mit einem anderen Tutorial das Erstellen von Windows-Anwendungen erlernen, um sinnvolle Editoren und Tools (z.B. für Spiele) zu entwickeln. Anschließend solltest du dann dasderdie DirectX API erlernen, um dir Spiele zu erstellen.

Doch bevor du große Pläne schmiedest, was du als erstes für ein Spiel programmierst, musst du halt erst mal Programmieren lernen. Eine realistische Betrachtungsweise ist hier ganz wichtig!! (Kumpels und ich haben in einem vergangenem Jahr wochenlang geplant, Scanner gekauft, Bilder von mittelalterlichen Rüstungen gesucht und gescannt, bis ich die Kumpels überzeugen konnte, erst mal Programmieren zu lernen, dann erst Ressourcen zu machen - und auf einmal schien keiner mehr Lust darauf zu haben - mach so was nicht, es nimmt dir vielleicht die Lust und den Reiz am Programmieren - wir wollten keine Geduld üben, und sofort gewaltige Ergebnisse bekommen; ich suchte im Internet nach Spielprogrammier-Programme (sogenannte Autorensysteme) und fand letzten Endes zu einem einzigen Ergebnis: RICHTIG PROGRAMMIEREN LERNEN!).

WIE LANGE DAUERT DAS DENN SO???

Eine gute Zeit ist 10 bis 20 Wochen, wenn man Schüler ist. Studenten haben natürlich mehr um die Ohren und müssen auch damit rechnen, dass sie so wenig Zeit haben, dass sie das Gelernte wieder vergessen und deswegen noch mehr Zeit brauchen. 20 bis 40 Wochen würde ich so schätzen!? Arbeitslose haben sehr viel Freizeit und können ziemlich ungehindert agieren - etwa 7 bis 15 Wochen. Berufstätige würde ich wie die Studenten einschätzen. Wer Ferien oder Urlaub hat, kann diese Zeit gut dazu nutzen; es wäre aber nicht zu empfehlen, erst auf Ferien zu warten, oder zu versuchen, sich alles in der kurzen Zeit zu pressen!

Computerspiele verleiten einen Menschen immer gerne zu einem Viertelstündchen zocken (nebenbei wurden sie ja dafür gemacht) - die nicht widerstehen können, sollten sich gegebenenfalls brutal durchsetzen und die Haupt-Zock-Programme deinstallieren (man sollte wissen, ob man lieber Spielen spielt oder Spiele macht) - auch ich erwische mich ab und zu mal beim Diablo 2-zocken.

WAS MUSS ICH DENN SCHON WISSEN, WENN ICH ANFANGEN WILL???

Du solltest einiges über Windows (wo was ist; was einige Optionen bedeuten) und über Computer generell wissen (nicht irgendwelche Geschichte, eher was ein Prozessor und eine Festplatte ist, was das BIOS für eine Aufgabe hat...).

Wichtiger allerdings: Du musst Englisch können, da C++ und der Compiler Englisch sind, nebenbei ist Englisch eine Weltsprache und viele Webseiten, wo es Tutorials gibt, sind auch Englisch.

Außerdem solltest du dir alle Beispiele anschauen, Kompilieren (im folgenden compeilern genannt), hin und wieder absichtlich Fehler einbauen und Werte verändern und sehen, was passiert. So lernst du, mit Compiler-Fehlern umzugehen und was du bei diesen zu tun hast.

WAS BRAUCH ICH JETZT NOCH???

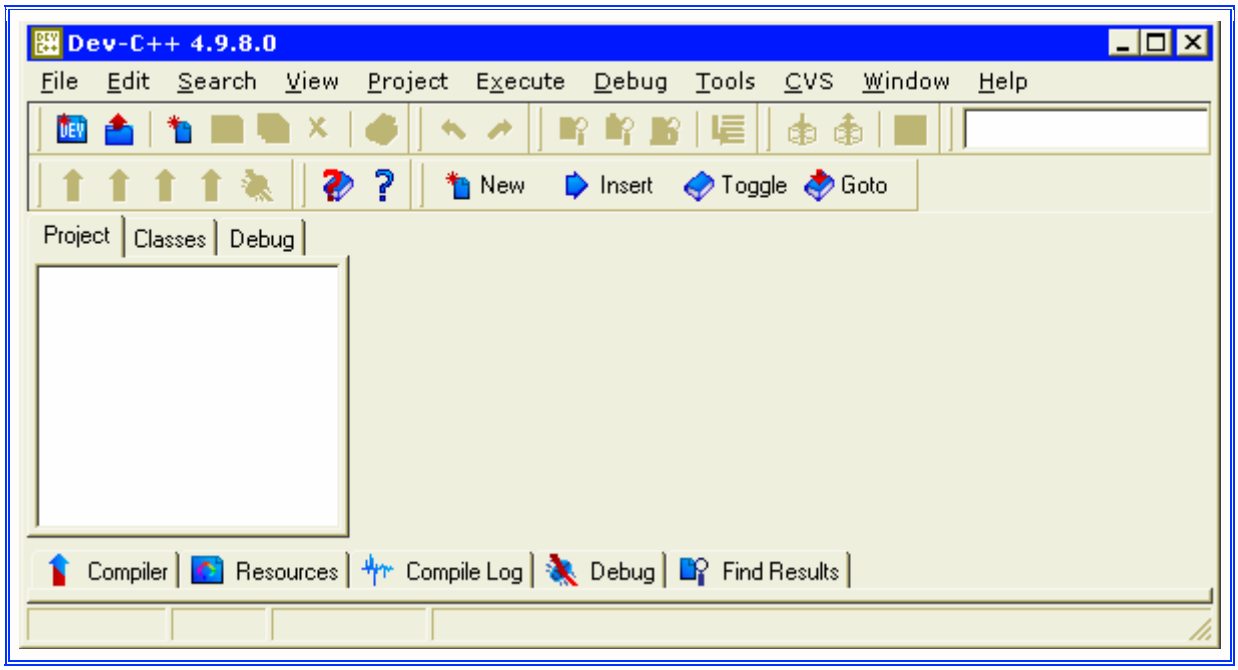
Fehlen tut jetzt noch das Programm, das den Code übersetzt: der schon mehrmals genannte Compiler!

Mein persönlicher Favorit: Dev-C++ - ein englischer Freewarecompiler, vom Bloodshed.net Team entwickelt. Gerade der Anfänger kann und will es sich nicht leisten, einen recht teuren Compiler wie MS Visual C++ oder den von Borland zu kaufen. Dazu gibt es sowieso keinen Grund: der Freewarecompiler reicht auch für DirectX. Du als Beginner brauchst die ganzen Funktionen eh nicht, DEV-C++ GENÜGT VOLLKOMMEN, ist meiner Meinung nach teilweise auch besser als so teure Produkte.

Natürlich gibt es noch weitere Gratiscompiler:

-LCC
-DJGPP
-Borland command line tools
-weitere, die ich nicht kenne

Allerdings hat Dev-C++ eine recht große Fangemeinde, zu der auch ich mich zähle. Hier ist ein Bild des Programms (zusammengeschrumpelt):



Du kannst es unter **www.bloodshed.net/devcpp.html** downloaden. Derzeit steht Version 4 und Version 5 Beta zum Download bereit, nimm Version 5.

Informationen zu Dev-C++:

Informationen zu Dev-C++ - Eine kleine Starthilfe

Begriffe

Die folgenden Begriffe stehen im Zusammenhang mit Dev-C++ oder sind allgemein wissenswert:

-IDE (Integrated Development Environment) - Entwicklungsumgebung oder grafische Oberfläche für einen Compiler
-GNU (rekursiv für GNU's not Unix) - Großangelegtes Projekt, dessen Ziel es ist, ein UNIX-ähnliches Betriebssystem zu programmieren
-GCC (GNU Compiler Collection) - Eine Sammlung von Compilern für verschiedene Programmiersprachen (unter anderen C und C++)
-GPL (GNU General Public License) - Lizenz um sicherzustellen, dass mit GCC compilierte Software für jeden benutzbar und auch änderbar ist ("Open Source")
-MinGW (Minimalist GNU for Windows) - Eine Sammlung von Headerdateien und Bibliotheken, mit denen man Programme für Windows schreiben kann (eine DLL-Datei von MinGW wird benötigt)
-CygWin (Cygnus and Windows) - Alternative zu MinGW (eine CygWin-DLL wird benötigt)

Dev-C++ ist eigentlich eine IDE für einen GCC Compiler (GCC kommt komplett ohne grafischen Elemente aus, deswegen entwickelt man IDEs). Zuerst modifiziert der Präprozessor noch den Quellcode. Der Compiler übersetzt dann den Quellcode in den sogenannten Objektcode. Diesen nimmt ein weiteres Programm, der Linker, und übersetzt ihn in Maschinen verständlichen Code - die Exe.

Man könnte das Übersetzen also in diese drei Vorgänge unterteilen. Der Einfachheit halber verwendet man sehr oft 'compilern' dafür; so wird auch das, was eigentlich die IDE ist, zum 'Compiler'.

> Mit dem gcc Version 3.2, der in Dev-C++ 5 integriert ist, bekommst du in diesem Tutorial immer ziemlich große Exen raus. Das kann natürlich gar nicht sein, vor allem wenn ein kleines Programm mit nur 10 oder 20 Zeilen kompiliert wird.
>
> Am Ende werde ich kurz auf Compiler-Optionen eingehen, die die Files etwas kleiner machen können und den Code optimieren. Damit solltest du dich später mal beschäftigen.

Installation von Dev-C++ 5

Die sollte nicht schwer fallen; halte dich möglichst an die Vorgaben (besonders der Installationspfad "C:\Dev-Cpp" - keine Leerzeichen in der Pfadangabe!!!).

Konfiguration der IDE

Wenn du das Programm das erste mal startest (bzw. es das selbst tut), wirst du mit einem Willkommensbildschirm begrüßt, in dem du einstellen kannst, welche Sprache, welche Icons du haben willst (die blauen hab ich gemacht :) und ob es in einem XP-Stil dargestellt werden soll. Ich empfehle als Sprache immer noch Englisch, die deutsche Version ist noch nicht perfekt. Dieses Tutorial bezieht sich

auf englische Namen.

Nach bestätigen dieses Dialogs erscheint das Hauptfenster. Klick den folgenden Link: **devcpp**. Wahrscheinlich erscheint der Dialog Öffnen-Speichern-Abbrechen-Details oder so. Damit es nicht allzu sehr nervt, checkst du "Vor dem Öffnen dieses Dateityps immer bestätigen" und klickst dann Öffnen.

Klicke im Projektbrowser (links) auf **devcpp1.cpp**. Daraufhin öffnet sich die Quellcodedatei, in der du einige Sprachelemente aufgelistet siehst. Hier siehst du ein paar Einstellungen, wie ich sie dir empfehlen würde:

Zeilennummern

- Menu "Tools" -> "Editor Options" -> Seite "Display"
- "Line Numbers" aktivieren
- OK

Damit wird für jede Zeile im Quellcode eine Zeilenangabe links gemacht. Dies ist sehr nützlich, denn es hilft, Überblick zu bewahren. Um diese Option gibt es noch weitere Einstellungen, die den linken Rand betreffen. Ich würde eine andere Schriftart wählen.

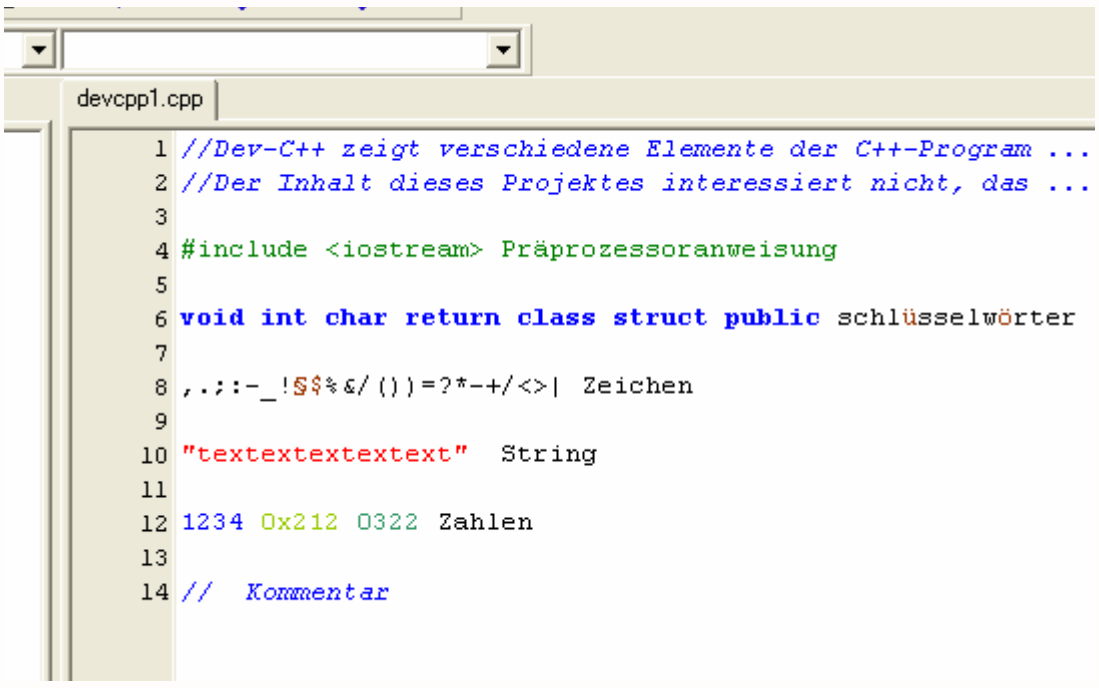
Editor

- Menu "Tools" -> "Editor Options" -> Seite "General"
- "Cursor Past EOL" aktivieren
- "Keep Trailing Spaces" deaktivieren
- "Tab Size" auf 3 einstellen
- OK

Darstellung der Sprachelemente

- Menu "Tools" -> "Editor Options" -> Seite "Syntax"

Hier kannst du einstellen, wie die Sprachelemente dargestellt werden sollen. So sieht das bei mir aus:

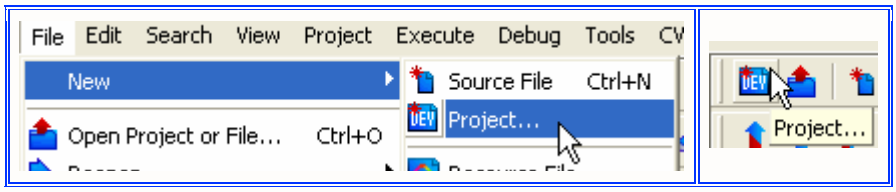


- ungültige Zeichen wie ü und ö und § und $ sofort kenntlich machen
- Zahlensysteme unterschiedlich färben

Benutzen von Dev-C++

Neues Projekt starten:

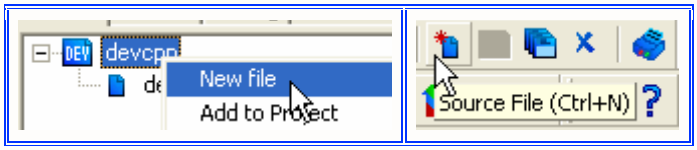
- Menu "File" -> "New" -> "Project".
- Oder in der Symbolleiste das Symbol, was dem im Menu "File" entspricht.



- Für ein DOS Projekt musst du "Console Application" wählen. Normalerweise ist C++ als Sprache gewählt, falls nicht, musst du es von C zu C++ umstellen. Projektnamen eingeben nicht vergessen.
- OK
- Speicherort angeben (sollte etwa C:\Dev-Cpp\Test sein).
- Das sich öffnende Fenster ist das Editorfenster, wo du deinen Code reinschreiben musst (haha).
- Den vorgegebenen Code kannst du löschen.

Quellcodedatei hinzufügen:

- Rechtsklick auf den Namen des Projekts im Projektmanager links.
- "New File" anwählen
- Speichern oder gleich Kompilieren (mit automatischen Speichern).



Kompilieren (im folgenden compeilern genannt) und Ausführen

- Menu "Execute" -> "Compile & Run"
- Oder selbiges Symbol in der Symbolleiste oder F9 drücken.

Für jedes Symbol in der Symbolleiste gibt es ein Symbol in den einzelnen Menus mitsamt Funktion. Viele der Funktionen haben einen Shortkey - F9 dürfte wohl der wichtigste sein!

Wie oben schon gesagt: Wenn du einen Link zu einem Projekt öffnen willst, fragt der Internet Explorer, ob du die Datei downloaden willst (in einem Dialog: Öffnen, Speichern, Abbrechen, Details). Du solltest das Kontrollkästchen "Öffnen von

Dateien dieses Typs immer bestätigen" deaktivieren, damit du nicht jedes Mal mit Öffnen bestätigen musst.

Den Rest musst du selber rausfinden. Tüftel nicht an den "Compiler Options" rum.

(Dieses Tutorial ist eines der leider Wenigen, das für seine Projekte den Compiler Dev-C++ wählt (andere nehmen entweder >ist-dir-überlassen< oder MS VisualC++). Dazu ist es selten, dass Tutorials komplette Code-Listings beinhalten. Noch seltener trifft man Tutorials mit kompletten Projekten für einen gewählten Compiler. Und ebenfalls selten sind Tutorials, die in ihrem HTML-Dasein die Möglichkeit nutzen, Links zu vorhandenen Projektdateien bereitzustellen. Das vereinfacht dir das Programmieren lernen sicherlich, da du komplette Programme siehst, birgt aber auch den Nachteil, dass du dir komplette Lösungen raussuchst und verwendest, ohne darüber nachzudenken und sozusagen am Zweck dieses Tutorials vorbeischlitterst. Aber das wurde ja schon im Essay oben genannt.)

Damit die Seiten optimal dargestellt werden können, sollte dein Bildschirm mindestens eine Auflösung von 1024x768 haben (am besten ist 1280x1024 geeignet). Als Browser empfehl ich den MS Internet Explorer 6, andere Browser könnten Probleme mit der Darstellung oder den Dev-C++ Projekten machen. Falls dir der Hintergrund nicht gefällt, kannst du "bg.gif" bearbeiten. Eine Änderung an diesem Bild wirkt sich auf alle Seiten gleichermaßen aus, solange es eine Datei "bg.gif" gibt. Die Datei "css_style.css" beinhaltet weitere Einstellungen, die über fast das gesamte Design dieses Kurses entscheiden - passt dir dieses grün hier nicht, kannst du es einfach umändern.

Hier gleich mal Links zu alternativen Tutorials:

http://www.schornboeck.net/ckurs/inhalt.htm
http://www.math.uni-wuppertal.de/~axel/skripte/oop/oop.html
http://www.volkard.de/vcppkold/inhalt.html

Die kannst du zur Hand nehmen, wenn du Sachen bei meinem Tutorial nicht gut oder falsch erklärt findest. In den einzelnen Kapiteln werde ich noch Links zu speziellen Kapiteln dieser Tutorials anbringen.

Bücher, die ich empfehle:

"C++ in 21 Tagen" - Jesse Liberty - ISBN: 3-8272-5624-1; Online-Version: **http://www.informit.de/books/c++21/data/inhalt.htm**
"C/C++ GE-PACKT" - Herbert Schildt - ISBN: 3-8266-0684-1
"OOP für Dummies" - Marcus Bäckmann - ISBN: 3-8266-2984-1

Kapitel 1

Nun mal los! - Die Hauptfunktion

Klick den folgenden Link an: main1

Dev-C++ sollte sich öffnen; falls jedoch ein Dialog des Internet-Explorers (Öffnen, Details, Abbrechen, Speichern) kommt, deaktiviere das Kontrollkästchen "Vor dem Öffnen dieses Dateityps immer bestätigen" und klicke dann Öffnen. Spätestens jetzt sollte sich Dev-C++ öffnen!

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

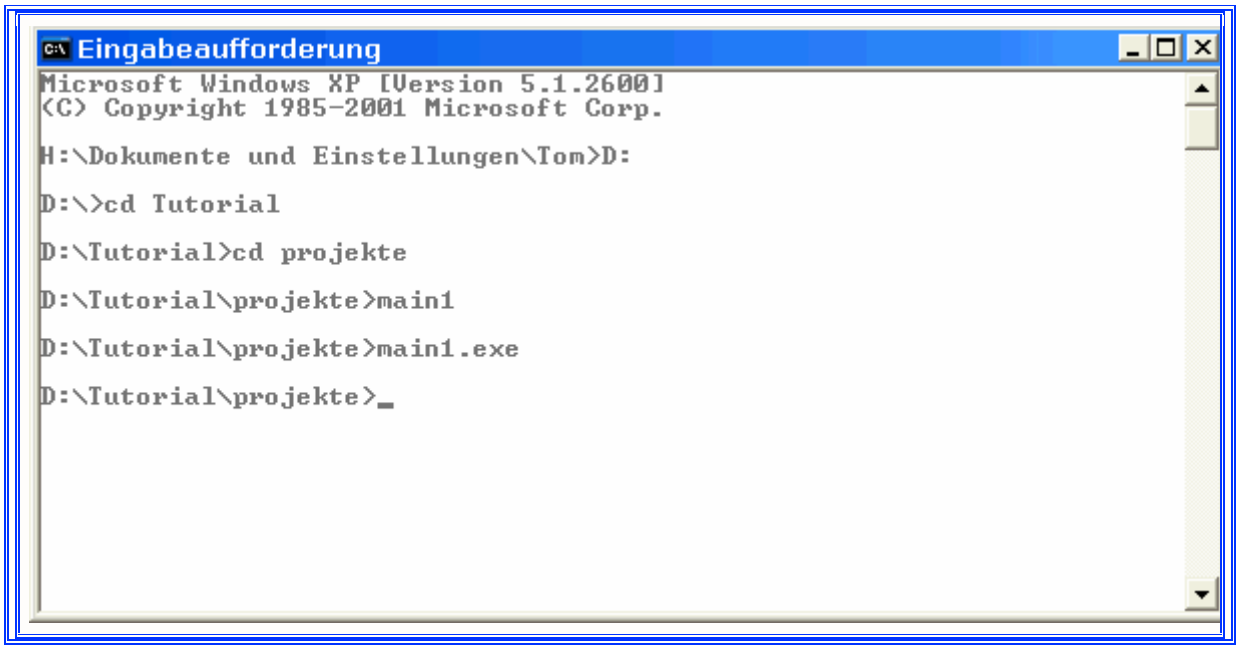
Das ist die `main`-Funktion - der Code darin wird bei einem DOS-Programm zuerst ausgeführt. Compeil das Programm und du wirst sehen, dass sich ein Fenster öffnet - unter Windows also eine MS-DOS Eingabeaufforderung - und auch gleich wieder schließt. Es fehlt an Inhalt!!

In diesem ersten Kapitel kannst du die Ausgaben auf den Bildschrim nur richtig sehen, wenn du das Programm von der Windows-eigenen DOS-Eingabeaufforderung aufrufst. Gleich im nächsten Kapitel (um noch etwas Spannung zu verbreiten) allerdings kommt eine bessere Möglichkeit hinzu, um das Fenster offen zu halten.

Auch wenn die Beispiele **main1** und **main2** keinerlei Zeichen ausgeben, zeige ich hier kurz, wie das in der Eingabeaufforderung aussehen kann:

Eingabeaufforderung unter Windows 98 und ME

Eingabeaufforderung unter Windows 2000 und XP



Die Wörter

`int` und **`void`** sind Schlüsselworter von C++, das heißt, dass es zum Sprachumfang von C++ gehört. Schlüsselwörter kann man in jedem Programm benutzen. Schlüsselwörter werden von vielen IDEs fett dargestellt, so auch Dev-C++. **`void`** heißt ins Deutsche übersetzt "nichts". **`int`** ist ein Datentyp, der Ganzzahlen speichert.

`main` ist kein Schlüsselwort - es stellt die Hauptfunktion eines jeden DOS-Programms dar. Schreibst du diesen Namen falsch, gibt Dev-C++ eine Linker-Fehlermeldung aus, und zwar "... undefined reference to 'WinMain@16' ". Merk dir:

C++ (und auch C) sind Case-sensitive, sie unterscheiden Groß- und Kleinschreibung, beispielsweise wäre **`void`** nicht `Void` oder `VoID` und `main` nicht `MAIN` oder `mAIn`. Der Compiler ist kleinlicher als jeder Deutschlehrer!!

Falsch wäre also: `Int Main(Void)` und `INT MAIN(VOID)`.

`return` `0` sollte in jede `main`-Funktion rein, die dem C++ Standard folgt. **`return`** ist ein weiteres Schlüsselwort und gibt einen Wert von der Funktion zurück - bei der `main`-Funktion normalerweise `0`. Wenn ein **`return`** vor ein paar anderen Anweisungen im Code steht, werden diese nicht mehr ausgeführt, weil es sozusagen die Kontrolle an die vorherige Funktion übergibt. (Im Programm ganz oben ist **`return`** `0;` die erste und einzige Anweisung, die die `main`-Funktion ausführt.)

Danach steht ein Semikolon - es drückt aus, dass ein Befehl zuende ist. Wie in einer realen Sprache musst du Satzzeichen setzen, sonst kann der Sinn des Satzes verfälscht werden. Bei **`return`** `0;` beendet es also den **`return`**-Befehl.

Syntax der main-Funktion

Wie schon erwähnt, ist `main()` die Hauptfunktion. Dabei handelt es sich um den für Funktionen typischen Syntax:

```
Rückgabetyp Funktionsname ( Parameter )
```

Doch dazu berichte ich in einem späteren Kapitel detaillierter. Der C++ Standard schreibt für die `main`-Funktion entweder den Syntax von oben oder einen Syntax, den ich später auch noch zeigen werde, vor.

Die geschweiften Klammern deuten an, dass darin der Code für die `main`-Funktion steht - derzeit nur die Rückgabe mit **`return`**. Jedes DOS-Programm muss eine `main`-Funktion enthalten, also darfst du die Hauptfunktion auch nicht umbenennen (bzw. falsch schreiben).

Im Kapitel über Funktionen wirst du noch eine zweite Variante sehen, der Vollständigkeit halber kurz mal gezeigt:

```
int main(int argc, char *argv[])
{
   return 0;
}
```

Kommentare

Nächstes Projekt - main2

Selbe Funktion, nur mit etwas mehr Inhalt. Der ist allerdings nicht für den Compiler interessant:

Kommentare

In Zeile 4 wird ein einzeiliges Kommentar verwendet. Ein Kommentar kannst du nach jedem Befehl gebrauchen, es geht jedoch auch in einer Extrazeile. Der Compiler ignoriert es, denn es dient dem Programmierer als Notiz.

`//` - einzeilig
`/*` - Anfang eines mehrzeiligen
`*/` - Ende eines mehrzeiligen

Kommentare kannst du fast überall im Code ablassen.

Verwendungszwecke:

1.)

Kommentare dienen natürlich der Verständlichkeit und Lesbarkeit des Codes. Allerdings immer nur in dem Maße, dass sie entweder ans Ende der Zeile oder in einen eigenen Absatz kommen. Du kannst zwar davon ausgehen, dass man deinen Code auch ohne Kommentare lesen und verstehen kann, dann ist aber deine Absicht für den Leser wahrscheinlich nicht klar und er wird wesentlich länger brauchen als mit angebrachten Kommentaren.

Was zu kommentieren ist und was nicht, wird beim Coden meistens erkenntlich. Korrektes Kommentieren lässt sich recht einfach lernen (wenn das überhaupt nötig ist). Hier ein paar Regeln für "gutes" kommentieren (du solltest später noch einmal hierher kommen, damit du weißt, was die Begriffe bedeuten):

- selten oder temporär gebrauchte Variablen mit einem Kommentar erkenntlich machen

- bei Funktionsprototypen die Verwendung der Rückgabewerte und Parameter nennen
- größere Sinnabschnitte gliedern und mit Kommentaren beginnen lassen
- Module einen Anfangskommentar verpassen ("Comment Header")
- Fehler mit einem langen `/////////` markieren
- bei Klassen und Funktionen den Programmierer und den Programmierstatus angeben
- Kommentare immer auf dem aktuellsten Stand bringen

Nicht zu empfehlen ist übermäßiges Kommentieren - einfache Anweisungen sind unkommentiert zu lassen! Die Übersichtlichkeit leidet enorm, wenn du jeder Zeile (oder jeder zweiten oder dritten) ein Kommentar anhängst.

2.)

Nicht zu vergessen und sehr nützlich ist, Befehle vom Compilern auszuschließen und somit Code-Fragmente zu "konservieren". Ein Stückchen alter Code kann also weiterhin neben einem neuen alternativen Code stehen. Alte Ideen bleiben erhalten und können einem neuen Lösungsweg möglicherweise helfen.

Ein Fallstrick bei Kommentaren ist allerdings, mehrzeilige Kommentare zu schachteln:

```
int main(void)
{
   /*außen /*innen*/ (wieder außen - Fehler)*/
   //...
}
```

Der Kommentar endet schon nach "`innen`"! Auch ein simples `*/` erzeugt einen Fehler! Du musst also auf sowas achten, vor allem bei auskommentiertem Code.

Das Ausgabe-Objekt cout

Ausgabe mit cout - cout1

Beim Ausführen siehst du kurz das DOS-Fenster, und kannst vielleicht sogar erkennen, dass da was geschrieben steht, nämlich "`Ausgabe mit cout`". Du solltest jetzt mal die MS DOS Eingabeaufforderung verwenden, damit du den Inhalt nachlesen kannst (falls dieser nicht klar sein sollte). Der `cout`-Befehl gibt Text auf den Bildschirm aus - hier Strings (Zeichenketten), später noch anderes. Diese müssen in 2 Anführungszeichen stehen:

```
cout << "Text, Zahlen und Kram";
```

Du kannst auch schreiben:

```
cout << "Aus" << "gabe" << "mit c"<< "out";
```

Du kannst zwischen einem `""` und einem `<<` auch eine neue Zeile beginnen:

```
cout << "Te"
     << "xt";
```

Dann musst du das Semikolon hinter den letzten Stringfragment setzen.

Genauer erklärt:

> `cout` ist kein Schlüsselwort, sondern ein Ausgabeobjekt - ein Objekt der objektorientierten Programmierung zur Ausgabe von Daten auf den Bildschirm. Du wirst später was darüber erfahren - wichtig ist, dass du damit komfortabel Daten ausgeben kannst.
>
> Noch zwei Neuerungen gegenüber dem vorigem Programm (welches **main2** war):
>
> Die Anweisung `#include <iostream>` :
>
> > `#` - Eine Raute veranlasst den Präprozessor, die nachfolgende Anweisung auszuführen. Der Präprozessor ist ein Programm, dass noch vor dem compeilern abläuft. Präprozessor-Anweisungen werden in Dev-C++ normalerweise grün dargestellt.
> >
> > `include` - zu deutsch "Einbeziehen, inkludieren". Der Präprozessor ersetzt die gesamte Anweisung durch den Inhalt der einbezogenen Datei. Dateien in 2 eckigen Klammern `<>` sucht der Compiler im Standard-Include-Verzeichniss - **C:\Dev-Cpp\include** . Dateien in Anführungszeichen `""` sucht der Compiler in dem Verzeichnis, wo das Projekt gespeichert ist (kommt noch im Kapitel "modulare Programmierung").

Kleine Datenkunde

***.dev** - Dev-C++ Projektdatei, kann nur von Dev-C++ geöffnet werden
***.c** - C Quellcodedatei
***.h** - C und C++ Headerdatei
***.cpp** - C++ Quellcodedatei
***.hpp** - C++ Headerdatei
***** - ohne Endung - C++ Standard-Headerdatei (nach neuem C++ Standard)

Typischerweise verwendet man aber, obwohl C++ eigene ***.hpp** Dateien hat, meist C-typische ***.h** Header. Headerdateien soll uns aber erst später interessieren.

> > `#include <iostream>` heißt also: **iostream** in dem Standard-Include-Verzeichniss suchen und dann die Anweisung `#include <iostream>` durch den Code, der in der **iostream**-Datei steht, ersetzen. **iostream** beinhaltet wichtige Funktionen (bzw. Objekte) zur Daten-Aus- und Eingabe, darunter `cout`.
>
> Die Anweisung **`using namespace std;`**:
>
> > In den C++ Headerdateien gehören sämtliche Funktionen und Objekte dem sogenannten Namensraum (**namespace**) `std` an. Würde die Anweisung nicht dastehen, müsstest du alle Objekte und Funktionen des jeweiligen Headers mit `std::FunktionOderObjekt` ansprechen. Die Anweisung gibt also bekannt, dass dieser spezielle Namensraum allgemeingültig ist.

Du wirst in fast jedem Beispielprojekt diese Anweisung finden, weil viele der Standardheader ihre Funktionen und Objekte in diesen Namensraum gepackt haben und dieses `std::` auf Dauer stört.

Ein Wort zur Formatierung

Beispiel format1.

Um Code übersichtlich zu halten, nutzt du Leerzeichen, Tabulator und Enter. Je besser Quellcodedateien formatiert sind, desto besser kannst du sie später lesen. Bei sehr langen Ausdrücken in einer Zeile solltest du versuchen, diese in mehrere Zeilen aufzuspalten. Hier einige Tips:

- jede Anweisung in eine eigene Zeile
- nach jeder geschweiften Klammer mit 3 Leerzeichen einrücken
- den Code klar in Abschnitte gliedern (Sinnabschnitte)
- lange Ausdrücke möglichst so kurz halten, dass man nicht quer scrollen muss (an den grauen Strich, der in Dev-C++ angezeigt wird, halten - der ist auf 80 Zeichen eingestellt)

Zusammenfassung

Dieses Kapitel war noch sehr einfach, schließlich waren es nur die grundsätzlichen Sachen. Jetzt weißt du, was die `main`-Funktion ist, wozu man sie braucht und so weiter. Merk dir, dass jedes C++ Programm zwingend EINE `main`-Funktion haben muss, und zwar kann diese laut Standard in zwei Formen vorliegen (mit und ohne Parameter).

Du hast `cout` und die dafür erforderlichen Vorbereitungen kennengelernt. `cout` lässt dich komfortabel Texte ausgeben. Die Vorbereitungen sind den **iostream**-Header einzubinden und den `std`-Namensraum in den globalen Namensraum aufzunehmen.

Weitere wichtige Punkte in diesem Kapitel waren Kommentare und die Formatierung des Codes. Beide dienen der Lesbarkeit und Übersichtlichkeit.

Wenn du Probleme mit dieser Einführung hattest, dann nimm dir eben noch einmal die Zeit, alles durchzulesen.

Workshop

Zeit zum üben

1.) Was ist denn eine `main`-Funktion??
2.) Wo kommt der Code der `main`-Funktion rein??
3.) Was bedeutet dieses Semikolon hinter dem `cout`-Befehl??
4.) Was macht die `cout`-Funktion??
5.) Wie kann man lange Strings, die nicht auf einmal auf eine Breite der Quellcode-Anzeige passen, auf mehrere Zeilen verteilen??
6.) Was empfiehlt sich hinsichtlich der Lesbarkeit des Codes zu unternehmen??
7.) Was schreibt uns der Standard für C++-Programme vor??
8.) Kann es zwei `main`-Funktionen im gleichen Projekt geben??

Programme

1.) Schreibe ein Programm, das deinen Namen ausgibt, davor jedoch die Welt begrüßt - "`Hello, World`" !

2.) Das gleiche, allerdings auf mehrere Zeilen in der cpp-Datei aufgeteilt!

Bevor du dir die Lösungen am Computer anschaust, probier erst mal alles aus, was du weißt. Eigentlich sind Lösungen hier unnötig, denn das alles ist leicht zu verstehen.

Links:

http://www.volkard.de/vcppkold/die_erste_anwendung.html - muss an neuen Standard angepasst werden!!
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop2_2.html
http://www.schornboeck.net/ckurs/hallo_welt.htm

Kapitel 2

Was wäre ein Programm ohne den Einsatz von Speicher?? In diesem Abschnitt will ich dir beibringen, wie Variablen und Operatoren in C++ einzusetzen sind.

Dazu machen wir einen kurzen Ausflug in den Computer:

Speicher

Speicher gibt es in absolut jedem Computer. Er dient natürlich dazu, sich Daten/Informationen zu merken. Variablen (und ein Teil von Konstanten) stellen solche Daten/Informationen dar. Dazu legt der Computer an einem bestimmten Ort - der Adresse - diese Informationen ab. Ein Programm mit Variablen muss sich zwangsläufig darum kümmern, dass das System bzw. das Betriebssystem ihm den erforderlichen Speicher zur Verfügung stellt.

Im Computer spielen zwei Zustände die Hauptrolle: 1 und 0! 1 für Stromfluss und 0 für keinen Stromfluss. Diese beiden Werte bilden ein Zahlensystem, das mit dem Dezimalzahlensystem vergleichbar ist - das Binärsystem. Der Speicher besteht also aus lauter Nullen und Einsen.

Jede Speicherstelle für eine Null oder eine Eins ist ein Bit. 8 Bit sind zusammen ein Byte. Ein Byte kann 2 hoch 8 Werte haben, also 0 bis 255 (insgesamt sind das 256 Werte). Als Beispiel nehm ich die Dezimalzahl **234**:

dezimal	$2 \cdot 100 + 3 \cdot 10 + 4 \cdot 1 =$ $2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 =$ **234** im Dezimalsystem
binär	$1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 =$ $1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$ **11101010** im Binärsystem

Einige Zahlenpaare:

dezimal	binär (1 Byte)
1	**00000001**
2	**00000010**
10	**00001010**
99	**01100011**
127	**01111111**
255	**11111111**

Binär **00000000** entspricht dezimal **0** und binär **11111111** ist dezimal **255**. Natürlich gibt es auch Variablen mit mehr als 8 Bit/1 Byte: ein 2-Byte Typ kann 2^{16} Werte und ein 4-Byte Typ 2^{32} Werte beinhalten. Es gibt auch 8-Byte Typen oder noch mehr - die sind aber in diesem Tutorial unwichtig und werden eigentlich nur in Spezialfällen zum Einsatz kommen.

Ein Computer kann immer nur so viel Speicher adressieren - d.h. ansprechen - wie er konzipiert ist. Ein 16-Bit System/Betriebssystem kann nur 65536 Byte adressieren - gerade mal ~65 KByte. Mit 32 Bit-Systemen ist schon einiges mehr möglich, denn ein solcher Computer kann bis 4294967296 Byte

ansprechen - das entspricht 4 Gigabyte Speicher. Oft sind Mainboards aber auf 2 oder 3 Gigabyte begrenzt wegen fehlender RAM-Slots. Kommende bzw. derzeitige 64-Bit-Monsterrechner könnten theoretisch bis 2^{64} Byte, also 18446744073709551616 Byte oder 16 Terabyte ansprechen!!!

Die Adresse kann so angegeben werden (auf ein 32 Bit-System bezogen): An Speicherstelle **01010101 10101010 00110011 01101100** ist der Wert **01101011** gespeichert. Das ist ein bischen umständlich und deswegen nimmt man einfach andere Zahlensysteme. Für Adressen hat sich das Hexadezimalsystem etabliert und für die Werte nimmt man das Dezimalsystem. In diesem Beispiel würde an der Adresse **55 AA 33 6C** der Wert **107** stehen.

Das Hexadezimalsystem hat 16 als Basis. Da es aber nur 10 Ziffern gibt, müssen 6 Buchstaben herhalten. Nachdem man also von 0 bis 9 gezählt hat, fährt man mit A B C D E F fort. Es lassen sich mit weniger "Ziffern" größere Zahlen als mit dem Dezimalsystem darstellen. Noch ein Beispiel mit der **234**:

dezimal	$2 \cdot 100 + 3 \cdot 10 + 4 \cdot 1 =$ $2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 =$ **234** im Dezimalsystem
hexadezimal	$0 \cdot 16^3 + 0 \cdot 256 + 14 \cdot 16 + 10 \cdot 1 =$ $0 \cdot 16^3 + 0 \cdot 16^2 + E \cdot 16 + A \cdot 1 =$ **EA** im Hexadezimalsystem

Auch hier wieder einige Zahlenpaare:

dezimal	hexadezimal (1 Byte)
1	**01**
2	**02**
10	**0A**
99	**63**
127	**87**
255	**FF**

Ein weiteres Zahlensystem, das es unweigerlich in der Informatik gibt, ist das Oktalsystem mit der Basis 8. Mögliche Ziffern sind dafür 0 1 2 3 4 5 6 7. Das Oktalsystem ist nicht so stark verbreitet, zumindest wird es beim ASCII-Code mit verwendet.

Dieser Abschnitt dient dem besseren Verständnis für Variablen und später für Zeiger.

Variablen

Eine Variable ist ein Platzhalter für einen Wert, den man ändern kann. Erstellt man in C++ eine Variable, so wird der dafür benötigte Speicherplatz im Arbeitsspeicher reserviert. Wenn man dieser Variable dann einen Wert zuweisen möchte, schreibt das Programm den Wert an die reservierte Stelle im Speicher. Das Erstellen bzw. Bekanntgeben einer Variablen heißt "Deklarieren". Wenn man gleich bei der Deklaration der Variablen einen Wert zuweist, nennt man das "Initialisieren" (weist man später einen Wert zu, so heißt es einfach "Zuweisen").

Variablen deklarieren

Der Syntax einer Variablendeklaration sieht jedes mal so aus:

```
Variablentyp Name ;
```

Beidesmal gibt es viele Möglichkeiten, nun erstmal zu den:

Variablentypen

C++ stellt dir verschiedene Grunddatentypen zur Verfügung. Eine kurze Übersicht siehst du hier:

Datentyp	Größe in Byte	Wertebereich
bool	1	0 oder 1 bzw. **true** oder **false**
char	1	-128 bis 127 - jeder Wert steht für das entsprechende Zeichen im ASCII-Code (-128 wird mit dem ASCII-Zeichen 0 verknüpft; 127 entspricht ASCII 255)
short	2	-32768 bis 32767
wchar_t	2	-32768 bis 32767 - ein 16-Bit **char**
int	2 oder 4	-32768 bis 32767 oder -2147483648 bis 2147483647
long	4	-2147483648 bis 2147483647
float	4	3.4E +/- 38 (7 Ziffern nach Komma)
double	8	1.7E +/- 308 (15 Ziffern nach Komma)
long double	10	1.2E +/- 4932

1 Byte ist eine Zusammensetzung aus 8 Bit - ein Bit kann den Wert `0` (**false**) und `1` (**true**) annehmen. Der Computer interpretiert ein Byte (Ziffernfolge aus 8 Nullen oder Einsen) nach dem Zweiersystem - näheres dazu hast du oben im Abschnitt **Speicher** gelesen.

bool ist ein Typ, der nur 2 Werte speichern kann - `0` (**false**) und `1` (**true**). Die Schlüsselwörter **true** und **false** sind extra für diesen Datentyp geschaffen.

char kann 256 (2^8) verschiedene Werte annehmen, die den im ASCII-Code definierten Zeichen entsprechen. **ASCII-Tabelle**.

wchar_t kann 65536 verschiedene Werte haben, die den im ANSI-Code definierten Zeichen entsprechen. Nur einige Windows-Schriftarten haben für jeden Wert ein Zeichen. Darin sind viele nicht-arabische Alphabete zu finden.

short kann 65536 verschiedene Zahlen annehmen - 32768 Negative und 32767 Positive und natürlich noch Null. **short** ist dasselbe wie **short int**.

int - ebenfalls für Ganzzahlen - kann neben Null auch 2147483648 negative und 2147483647 positive Werte annehmen, wenn die Variable in einer 32-bit Umgebung ist - z.B. Windows ab Version 95. Unter Dos und Windows 3.11 ist ein **int** 2 Byte groß, weil diese 16-bit-Umgebungen sind. Da Dev-C++ mit MinGW nur unter Windows ab Version 95 verfügbar ist, kannst du sicher sein, das sich **int** wie **long** verhält.

long ist auch für Ganzzahlen, und zwar im Bereich von -2147483648 bis 2147483647. Statt **long** kannst du auch **long int** schreiben.

Bevor es mit Beispielen losgehen kann, muss ich noch etwas erklären, und zwar die

Operatoren

Alle der folgenden Operatoren bis auf den Modulo-Operator kannst du auf Variablen der eben vorgestellten Datentypen anwenden. C++ hat sich nicht pingelig, auch wenn du mal **char** mit **float** multiplizierst :)

Für das Dezimalsystem:

Variablen initialisieren und Werte zuweisen

Dazu braucht man den Zuweisungsoperator =. Er bewirkt, wie du dir das denken kannst, dass der Ausdruck auf der rechten Seite der Variablen auf der linken Seite zugewiesen wird. Nehmen wir als Beispiel:

```
int VariableEins = 5; //Initialisierung

VariableEins = 128;   //Zuweisung
```

ist gültig, der Wert von VariableEins ist nun 128 geworden, nachdem er den Anfangswert 5 hatte. Foll valsch ist aber folgender Ausdruck:

```
128 = VariableEins;
```

,weil 128 kein gültiger Name für eine Variable ist; dazu kannst du später etwas lesen.

Du kannst auch mehreren Variablen ein und denselben Wert auf einmal zuweisen.

```
VariableEins = VariableZwei = VariableDrei = VariableVier;
```

Der Wert von VariableVier wird den Variablen VariableEins bis -Drei zugewiesen.

Arithmetische Operatoren

Es gibt neben den vier Grundrechenarten +-*/ auch noch den Rest von einer ganzzahligen Division - der Modulo-Operator.

```
+ für Addition
- für Subtraktion
* für Multiplikation
/ für Division

% als Modulo-Operator
```

Beispiele zu den Grundrechenarten:

```
VariableEins = 33+  3; // ergibt 36
VariableEins = 33 - 3; // 30
VariableEins = 33  *3; // 99
VariableEins = 33 / 3; // 11
```

Der rechte Ausdruck wird zuerst vollkommen ausgerechnet, dann dem linken Ausdruck zugewiesen. Du kannst Leerzeichen hinpacken wie du willst.

Den Modulo-Operator kann man nur auf Ganzzahlen - **short**, **int**, **long** - anwenden. Er liefert den Rest einer ganzzahligen Division. `41 / 4` wäre als Fließkommazahl `10.25` und als Ganzzahl `10` Rest `1`. Der Modulo-Operator gibt also `1`.

```
VariableEins = 41 % 4;  // 1
VariableEins = 99 % 10; // 9
```

Erweiterte Zuweisungsoperatoren

Wenn eine Variable verändert wird und das Ergebnis der Variable wieder zugewiesen wird, kannst du die erweiterten Zuweisungsoperatoren verwenden. Es gibt:

```
VariableEins += 10; // VariableEins = VariableEins + 10;
VariableEins -= 10; // VariableEins = VariableEins - 10;
VariableEins *= 10; // VariableEins = VariableEins * 10;
VariableEins /= 10; // VariableEins = VariableEins / 10;
```

Diese machen den Code natürlich ein bischen übersichtlicher.

Inkrement- und Dekrementoperatoren

Die stellen eine weitere Vereinfachung dar. Später wird es oft vorkommen, dass eine Variable um den Wert 1 erhöht oder verringert werden soll. Und genau das tun diese Operatoren:

```
VariableEins++; // Inkrementoperator
VariableEins--; // Dekrementoperator
```

Der Anhang `++` bedeutet dasselbe wie

```
VariableEins = VariableEins + 1;
```

Der Anhang `--` bedeutet dasselbe wie

```
VariableEins = VariableEins – 1;
```

Präfix und Postfix bei diesen Operatoren

Das war jetzt der Postfix. Es gibt noch den Präfix - beide spielen bei Zuweisungen eine Rolle:

```
VariableEins = VariableZwei++;
```
Postfix, `VariableEins` wird der Wert von `VariableZwei` zugewiesen, die dann um eins erhöht wird

```
VariableEins = VariableZwei--;
```
Postfix, `VariableEins` wird der Wert von `VariableZwei` zugewiesen, die dann um eins verringert wird

```
VariableEins = ++VariableZwei;
```
Präfix, `VariableZwei` wird um eins erhöht, und dann `VariableEins` zugewiesen

```
VariableEins = --VariableZwei;
```
Präfix, `VariableZwei` wird um eins erniedrigt, und dann `VariableEins` zugewiesen

Besserer Programmierstil ist aber folgender:

```
//Postfix

VariableEins = VariableZwei;
VariableZwei++;

VariableEins = VariableZwei;
VariableZwei--;
```

bzw.

```
//Präfix

VariableZwei++;
VariableEins = VariableZwei;

VariableZwei--;
VariableEins = VariableZwei;
```

Für das Binärsystem:

Variablen kannst du auch auf Binärbasis ändern. Computer können Bitoperatoren schneller verarbeiten als die arithmetischen Kollegen. Hier die vier Operatoren:

```
& als bitweises UND
| als bitweises ODER
^ als bitweises EXKLUSIV-ODER
~ als bitweises NICHT
```

Mit dem bitweisen UND-Operator kann man zwei Werte so verknüpfen, dass deren Ergebnis nur die Bits auf Eins lässt, die auch in den Ausgangswerten auf Eins standen:

Dezimal	sieht binär so aus:
5 & 4 = 4	00000101 & 00000100 = 00000100
101 & 31 = 100	01100101 & 00011111 = 00000101

Der ODER-Operator lässt nur da Einsen stehen, wo mindestens eine Eins in den Ausgangswerten stand.

Dezimal	sieht binär so aus:
5 | 4 = 5	00000101 | 00000100 = 00000101
101 | 31 = 127	01100101 | 00011111 = 01111111

Beim EXKLUSIV-ODER-Operator ist nur dann die Ergebnisstelle eine Eins, wenn sich die beiden Binärzahlen unterscheiden:

Dezimal	sieht binär so aus:
5 ^ 4 = 1	00000101 ^ 00000100 = 00000001
101 ^ 31 = 122	01100101 ^ 00011111 = 01111010

Beim NICHT-Operator wird nur eine Variable verändert, und zwar so, dass Nullen zu Einsen und Einsen zu Nullen werden:

Dezimal	sieht binär so aus:
~5 = 250	~00000101 = 11111010
~101 = 154	~01100101 = 10011010

bzw.

Dezimal	sieht binär so aus:
~4 = 251	~00000100 = 11111011
~31 = 224	~00011111 = 11100000

Des Weiteren gibt es Schiebeoperatoren, mit ihnen kann man alle Einsen im Binärwert der Zahl um einen beliebigen Betrag verschieben:

```
<< für das Verschieben der Einsen nach links
>> für das Verschieben der Einsen nach rechts
```

Dezimal	sieht binär so aus:
5 << 4 = 80	00000101 << 4 = 01010000
5 >> 2 = 1	00000101 >> 2 = 00000001
101 >> 2 = 25	01100101 >> 2 = 00011001
60 << 3 = 224	00111100 << 3 = 11100000

Es werden dabei jeweils Nullen aufgefüllt.

Anwenden kannst du das, indem du den veränderten Wert einer Variable zuweist:

```
VariableEins = VariableZwei | VariableDrei;
VariableEins = VariableZwei & VariableDrei;
VariableEins = VariableZwei ^ VariableDrei;
VariableEins = ~VariableDrei;
VariableEins = VariableZwei << VariableDrei;
VariableEins = VariableZwei >> VariableDrei;
```

Hier funktionieren auch die erweiterten Zuweisungsoperatoren:

```
VariableEins |= 33;
VariableEins &= 33;
VariableEins ^= 33;
VariableEins <<= 3;
VariableEins >>= 3;
```

Klammern (für beide Zahlensysteme)

Mit Klammern kannst du die Rangfolge in Ausdrücken gewollt verändern:

```
Variable = 25 * 2 + 2 –10; // 42
Variable = 25 * (2+2) -10; // 90
```

Operatoren für das Potenzieren oder Wurzeln gibt es in C++ nicht. Dafür werden wir später Funktionen kennenlernen.

Damit du deine Tastatur in deinen Programmen gebrauchen kannst:

Eingabe mit cin

Während du mit `cout <<` Informationen ausgeben lassen kannst, ist `cin >>` dafür gedacht, Variablen mit Daten von der Tastatur zu belegen. `cin` verlangt allerdings keine feststehende Zeichenkette, sondern eine Variable. Folgender Code würde eine Eingabe verlangen:

```
int Variable;
cin >> Variable;
```

Hier müsstest du eine ganz normale Dezimalzahl eingeben (nicht außerhalb des vorgesehenen Wertebereichs und auch kein Zeichen) und danach Enter drücken. Das wird uns in zukünftigen Programmen sehr nützlich sein, denn ohne Benutzereingabe ist fast jedes Programm armseelig. Zunächst soll eine solche Eingabe verhindern, dass das Programm sofort wieder schließt. Mit dieser sogenannten Dummy-Eingabe kannst du ja die Beispiele aus dem vorigen Kapitel bereichern, wenn du die Ausgaben mit `cout` sehen willst.

So, jetzt geht's mal kräftig los mit dem Code:

Ganzzahlen

Beispielprogramm variable1 zeigt, wie du Ganzzahlen deklarieren und initialisieren kannst.

Wenn du einer Variablen etwa des Typs **`short`** den Wert `600000` zuweist, ihn anschließend ausgeben lässt, erhältst du den Wert `10176`. Das lässt sich erklären: Du hast den Wertebereich überschritten. Ist der Wert 1 größer (bzw. kleiner) als der Wertebereich es zulässt, beginnt der Computer vom negativ (bzw. positiv) größten Wert weiterzuzählen, was bedeutet, dass zur Variable solange Eins dazuaddiert wird, bis 600000 Einsen hinzuaddiert wurden. Falls die Variable größer als `32767` wird, wird der Wert von der Variable zu `-32768`.

Fließkommazahlen

Beispielprogramm variable2 zeigt, was bei Fließkommawerten anders ist.

`float` und **`double`**-Werte musst du also mit einem Punkt als Komma angeben (Bsp: `23.695564`). Wenn du feststehende **`float`**-Werte einsetzen willst, solltest du ein '`f`' hinter den wert schreiben, damit du dem Compiler dies ausdrücklich klarmachst (wie das Beispiel gezeigt hat). Ansonsten verwendet der Compiler diesen Wert als **`double`**-Konstante.

Außerdem musst du beachten, dass bei der Division von Fließkommawerten durch Ganzzahlen der Punkt mit hingeschrieben werden muss:

```
float Variable = 3.3f;
float Variable2 = Variable / 2.0f;
```

Wenn du diese Angabe vergisst, würde für `Variable2` `1.15f` herauskommen und dann ganzzahlig gerundet werden (und `1` übrigbleiben).

Zeichen

Beispielprogramm variable3 zeigt den Einsatz von **char**-Werten.

Hier siehst du, wie verschiedenen **char**-Variablen konstante Zeichen zugewiesen werden. Einzelne Zeichen müssen dabei in einfachen Anführungszeichen 'x' stehen. Dies hast du auch schon bei Ausgaben mit `cout << xyz << ' ';` gesehen.

Mit **char**-Variablen kannst du genauso rechnen wie mit Ganzzahlen. Das scheint zunächst etwas komisch, ist aber durchaus praktisch.

Dir wird aufgefallen sein, dass die Zeichen 0 bis 31 im ASCII-Code eine wesentlich andere Bedeutung haben als die Restlichen - sie sind Steuerzeichen. Jedes dieser besonderen Zeichen hat eine bestimmte Verwendung für das Computer-System.

Wenn du einer **char**-Variable einen solchen Wert Wert zuweisen willst, musst du das gewünschte Zeichen dementsprechend besonders eingeben. Die C++-Synonyme sind in der folgenden Tabelle zusammengefasst:

Steuerzeichen	Bedeutung	ASCII Code-Nummer
`\n`	Neue Zeile	10
`\b`	Linkes Zeichen löschen	8
`\f`	Neue Seite	12
`\r`	Kursor geht zum Anfang der Zeile zurück	13
`\a`	Signalton (beep)	7
`\"`	Doppeltes Anführungszeichen	34
`\'`	Einfaches Anführungszeichen	39
`\\`	Backslash	92
`\?`	Fragezeichen	63
`\t`	Tabulator(horizontal)	9
`\0nn`	nn als Oktalwert für ein ASCII-Zeichen	Oktal nn
`\xnn`	nn als Hexalwert für ein ASCII-Zeichen	Hexal nn

Besonders das `\n` Steuerzeichen wird von großer Bedeutung sein, da du damit auszugebende Texte viel besser formatieren kannst.

Beispielprogramm variable4 zeigt das.

Hier werden komplette Zeichenketten (Strings) ausgegeben. Diese gehören in doppelte Anführungszeichen "`String`".

wchar_t verwendet einen größtenteils anderen Zeichensatz als **char**. Zeichenketten aus **wchar_t** müssen besonders gekennzeichnet werden:

```
L"Zeichenkette"
```

Da die Verwendung von **wchar_t**-Variablen etwas sonderbar ist, werde ich diese Art von Variablen nicht weiter beschreiben. Ich verwende im folgenden nur noch **char**-Variablen für Zeichen.

Der **bool**-Datentyp ist jetzt noch nicht wichtig, er wird erst im Abschnitt Fallunterscheidung eine Rolle spielen.

Zahlensysteme mit cout und im Code

Du kannst Ganzzahlen auch anders als im Dezimalzahlensystem ausgeben lassen, indem du cout passende Informationen gibst:

- dec - dezimal
- okt - oktal
- hex - hexadezimal

Doch wohin genau??

```
int Variable;

cout << dec << Variable;    // Variable wird wie immer Dezimal
ausgegeben
cout << okt << Variable;    // Variable wird durch okt als Oktal-
Wert ausgegeben
cout << hex << Variable;    // Variable wird durch hex als
Hexadezimal-Wert ausgegeben

cout << hex << Variable << dec << Variable; // Variable wird zuerst
als Hex-Wert ausgegeben
// dec ist notwendig, weil sonst Variable ein Zweites mal
hexadezimal ausgegeben wird
```

Du kannst auch Variablen in unterschiedlichen Zahlensystemen in den Code eingeben, etwa oktal durch voranstellen einer NULL (0) oder hexadezimal durch voranstellen von NULL X (0x):

```
int Dezimal = 10;
int Oktal = 012;
int Hexadez = 0xa;
```

Bei der Ausgabe einer Hexadezimal- oder Oktalzahl wirst du feststellen, dass cout 0x bzw. 0 nicht mit ausgibt. Falls du das jedoch wünschst, musst du showbase als Manipulator für cout angeben, andernfalls noshowbase oder einfach die Voreinstellung nehmen. Das machst du genauso wie bei dec, okt und hex:

```
cout << showbase << hex << 255;
```

Bei Dezimalbrüchen kannst du zwischen den zwei Arten "festkomma" und "wissenschaftlich" unterscheiden. Dabei ist normalerweise die Festkommaschreibweise gewählt, wenn es jedoch zu viele Ziffern nach dem Komma gibt, stellt cout die Zahl in der wissenschaftlichen Exponentialschreibweise dar. Für Exponentiell musst du scientific und für Festkomma fixed angeben.

Wenn du solche Extrawürste machst, solltest du immer ein Kommentar hinzufügen, damit deine Absicht klar wird. Eine Null kann schließlich schnell verloren gehen!

Ende Teil 1 der Variablen

Hier endet die Einführung in den Themenbereich Variablen. Im nächsten Kapitel wirst du weiteres Wichtiges erfahren!

Zusammenfassung

In diesem Kapitel hast du viele Sachen kennengelernt, die für Programme unerlässlich sind. Ich habe dir einen Einblick in den Speicher gegeben und dir die verschiedenen Zahlensysteme nahegebracht. Du weißt nun, was und wozu Variablen sind, was bei einzelnen Datentypen zu beachten ist und wie du welche Operatoren verwenden kannst.

Du kannst Variablen deklarieren, initialisieren und ihnen einen Wert zuweisen. Mithilfe von arithmetischen Operatoren und Bit-Operatoren kannst du den Wert von Variablen verändern. Zudem sollte es dir keine großen Probleme bereiten, rechnerische Ausdrücke in C++ zu formulieren.

Wieder hast du einen Kapitel geschafft und bist vorwärts gekommen. Auch wenn ich hier viel Theorie reingepackt habe, solltest du alles klar nachvollziehen haben können. Trink ein Glas klares (!Wasser!) und mach mal ne Pause.

Workshop

Zeit für eine Kontrolle

1.) Welche Zahlensysteme sind in C++ bedeutsam??
2.) Was musst du bei der Division bei **`float`** bzw. **`double`** beachten??
3.) Was ergeben die folgenden Ausdrücke??

Definitionen: `char A = 'a'; int B = 2468; float C = 0.5f; double D = 1.00555;`				
`A / C`	`A * 1.5f`	`A * (255-A)`	`A / 3`	`3 / 0.1`
`B + C + D`	`B * C * D`	`B / 15`	`A*C + B*D`	`B*B*B`
`0x123 + 25`	`046 * 0x34`	`035 / C`	`0x345 + 'f'`	`B / 02468`
`C / 2`	`C / 2.0f`	`C * D`	`D / C`	`4 / 3`

Einige davon kannst du im Kopf rechnen. Um jedoch das genaue Verhalten des Compilers herauszufinden, solltest du ein Programm dazu erstellen.

4.) Was ist mit **`bool`** und **`wchar_t`** los, dass die hier in diesem Kapitel nicht wirklich erklärt worden sind??
5.) Wieso sollte man die Möglichkeit gebrauchen, Zahlen durch andere Zahlensysteme darzustellen??
6.) Gibt es Unterschiede zwischen `cout` und `cin` ??
7.) Wieso kam `cin` erst in diesem Kapitel und nicht schon im vorigen??
8.) Wieso gibt es so viele arithmetische Operatoren in C++??

Programme

1.) Schreibe ein Programm, das deinen Namen mit mehreren **`char`**–Variablen ausgibt
2.) Schreibe ein Programm, welches Zehn Zahlen (von 1 bis 10) hintereinander ausgibt. Dazu solltest du nur eine Variable verwenden, die du mit verschiedenen Operatoren erhöhst (also nicht nur `++` und `+=`)

Links:

http://www.volkard.de/vcppkold/variablen.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop3_1.html
http://www.schornboeck.net/ckurs/variablen.htm

Kapitel 3

Hier noch einiges zum Thema Variablen:

- Konstanten
- Benennung von Variablen/Konstanten
- Gültigkeitsbereiche
- Variablenzusätze

Konstanten

Konstanten sind Werte, die während der Laufzeit des Programmes nicht geändert werden können. Sie sind also voll und ganz vom Quellcode abhängig.

Es gibt zwei Unterarten, zwischen denen man unterscheiden muss: literale und symbolische Konstanten. Kurz gesagt sind literale Konstanten Werte, die direkt an der gewünschten Stelle stehen und deren Wert also sofort ersichtlich ist. Symbolische Konstanten sind Werte, die durch ein "Symbol", also einen Namen, repräsentiert werden. In gewisser Weise sind symbolische Konstanten wie Variablen.

Literale Konstanten

Diese haben wir in der Tat schon eine ganze Weile verwendet. Literale Konstanten sind Zahlen, Buchstaben oder Strings, die genau da stehen, wo sie gebraucht werden:

```
cout << "Hallo" << 2+3 << ' ' << 0xa2;
```

Ganz einfach zu erkennen - `"Hallo"`, `2+3`, `' '` und `0xa2` sind literale Konstanten.

Symbolische Konstanten

Hier versteckt sich ein Wert hinter einem Symbol, also einem Namen. Diese solltest du genauso wie Variablen benennen - also den Sinn kurz beschreiben. (Näheres zur Benennung von Variablen und Konstanten gleich nach den Konstanten!)

Beispielsweise wäre es durchaus sinnvoll, die Kosten für 100 m^2 Parkettboden anstatt mit

```
float Cost = 100 * 13.99f;
```

durch

```
float Cost = Area * PricePerM2;
```

zu errechnen. Denn dann weiß man gleich die Absicht und das macht durchaus ein Kommentar überflüssig.

Bei der Deklaration von solchen Symbolkonstanten gibt es zwei verschiedene Arten:

1.) Als Variable mit dem Zusatz **const**

Mit dem Schlüsselwort **const**, vor den Variablentyp gesetzt, kannst du eine Konstante erzeugen, die du bei der Deklaration sofort initialisieren musst:

```
const float PI = 3.14159f;
```

Konstanten, die du auf die Art definierst, sind sehr sicher, weil der Compiler zu große Werte an der richtigen Stelle als Fehler markiert.

2.) Als definierter Wert mit `#define`

Mit Hilfe der Präprozessor-Anweisung `#define` kannst du Konstanten definieren:

```
#define PI 3.14159;
```

Für mit `#define` definierte Konstanten wird kein Speicher im Arbeitsspeicher reserviert, weil der Präprozessor noch vor dem Compilieren alle Konstanten im Quellcode sucht und durch den Wert (bei `PI` ist der Wert `3.14159f;`) ersetzt. `#define` braucht kein Semikolon, es gehört also zu PI mit dazu! Das ermöglicht es dir, so etwas zu schreiben:

```
cout << PI
```

Alle beiden Möglichkeiten veranschaulichen das **Beispiel const1**.

Mit `#define` kannst du jedoch nicht nur für Konstanten nutzen, sondern auch für komplette Ausdrücke:

```
#define NL cout << '\n';

int main(void)
{
   NL NL NL
   NL NL NL
   NL NL NL

   return 0;
}
```

Der erste Ausdruck hinter `#define` (also `NL`) ist der Name, alle folgenden Wörter sind die definierte Konstante, es ist also egal, ob und wie viele Leerzeichen in der definierten Konstanten stehen. Damit du komplexe Befehle auf mehrere Zeilen aufspalten kannst, musst du ein Backslash '`\`' hinter das letzte Zeichen der jeweiligen Zeile der Anweisung schreiben:

```
#define NL co\
ut\
<<\
```

```
'\n'\
;
```

Dieser Ausdruck ist eigentlich der gleiche wie oben, nur dass er zerlegt wurde. Das `\` als Steuerzeichen im Zeichen `'\n'` verhält sich allerdings nicht so wie das Zerlegungs-`\` eine Zeile darüber, weil nach dem Zeichen noch weitere andere Zeichen kommen.

In früheren Versionen des Tutorials ist mir ein Fehler unterlaufen:

```
#define NL co\
           ut\
           <<\
           '\n'\
           ;
```

Wird NL "aufgerufen", meckert der Compiler, dass er co und ut nicht kennt. Wie auch, wenn es doch eigentlich cout heißen sollte?? Die restlichen Zeilen gehen allerdings klar, weil auch in der normalen Einzeiler-Version Leerzeichen vorkommen.

Wenn du also Konstanten, die Zahlen oder Buchstaben beinhalten, verwenden willst, solltest du (entweder literale oder) symbolische **`const`**-Konstanten einsetzen. Obwohl eine **`const`**-Konstante Speicher verbraucht, solltest du die Sicherheit nutzen.

Benennung von Variablen und Konstanten

Bisher hab ich alle Variablen `VariableEins`, `VariableZwei` usw. und alle Konstanten `KONSTANTE1` usw. benannt. In folgenden Kapiteln werde ich das nicht mehr so oft tun (es sei denn, mir fällt kein gescheiter Name ein). Man kann sich natürlich alle Variablennamen selbst ausdenken, man muss dabei darauf achten, dass man die Variablen

- nicht mit einem Unterstrich beginnen lässt '_'
- nicht nach Schlüsselwörtern von C++ benennt (etwa **`void`**)
- nicht mit Nummern am Anfang des Namens versieht (etwa `12DiBromEthan`)
- nicht so benennt, wie es in einem Header schon getan wurde (doppelt definieren)
- dem Zweck entsprechend benennt (etwa `AnzahlRinge`)
- möglichst englische Namen nimmt (`NumberRings`)

Bei Variablennamen, die eigentlich aus mehreren Wörtern bestehen, solltest du neue Wörter immer groß schreiben (etwa `DerErsteBuchstabeDesAlphabets`).

Konstanten solltest du immer groß schreiben (etwa `KONSTANTE1` oder `PI`), damit man sie auf den ersten Blick erkennt.

Lokal und Global

Sämtliche bisher verwendeten Variablen und Konstanten waren global - sie wurden am Anfang des Programms deklariert, noch bevor die `main`-Funktion begonnen hat. Man kann sie aber auch in der `main`-Funktion deklarieren.

Beispiel global1.

In Zeile 8 wird eine **int**-Variable deklariert. Für die main-Funktion ist sie lokal, für den Block jedoch global, weil sie auch nach dem Block gültig ist. Die Variable a1 ist total global, sie ist überall gültig. c1 gilt jedoch nur im Block.

Beispiel global2.

Hier wird der Bereichsoperator ::xyz gezeigt. Das Programm greift bei einer solchen Anweisung nicht auf das lokalste xyz, sondern auf ein globaleres xyz.

Du solltest möglichst andere Namen für Variablen mit unterschiedlichen Gültigkeitsbereichen wählen, da du sonst schnell durcheinander kommst.

Variablenzusätze

Neben **const** gibt es noch weitere Zusätze, die du vor den Variablentyp schreiben kannst. Jeweils gilt die Reihenfolge:

```
Zusatz Variablentyp Name;
```

unsigned

Dieses Schlüsselwort legt fest, dass bei Variablen der Typen **int**, **float**, **double**, **long**, **short**, **wchar_t** und **char** keine negativen Werte möglich sind. Dadurch verdoppelt sich der Wertebereich in positive Richtung. Eine **short**-Variable geht dadurch bis maximal 65535.

signed

Genau das Gegenteil von **unsigned** - Variablen können negativ sowie positiv sein. Generell gilt: selbst wenn man diesen Zusatz weglässt, ist die Variable vorzeichenbehaftet.

register

Mit diesem Zusatz wird die Variable im Prozessorregister gespeichert, was teilweise eine große Geschwindigkeitsverbesserung mit sich zieht.

volatile

bewirkt, dass die Variable vom Compiler NICHT optimiert wird. Grund ist, dass so Fehler verhindert werden, wenn Quellen außerhalb des Programms zugreifen. Soll zunächst noch keine Rolle spielen.

Variablentyp durch Variablenzusätze ersetzen

register und **volatile** müssen vor einem Typen stehen; **signed** und **unsigned** werden implizit als **int** angenommen, wenn der Typ nicht spezifiziert ist.

Beispiel implicit1.

Zum Überblick zeige ich hier eine komplette Tabelle mit **unsigned** und **signed**:

Datentyp	Größe in Byte	Wertebereich	Konstante
bool	1	0 oder 1 bzw. **true** oder **false**	`0 1 true false`
char	1	-128 bis 127 (0 ist ASCII 0; -1 ist ASCII 255)	`'a' 'b' 'c' '\n'`
signed char	1	wie **char**	`'a' 'b' 'c' '\n'`
unsigned char	1	0 bis 255 (0 entspricht ASCII 0 :)	`'a' 'b' 'c' '\n'`
short	2	-32768 bis 32767	`5 -10 0xff 084`
signed short	2	-32768 bis 32767	`5 -10 0xff 084`
unsigned short	2	0 bis 65335	`5u 0xffU 084u`
wchar_t	2	-32768 bis 32767 - ein 16-Bit char	`'a' 'b' 'c' '\n'`
int	2 oder 4	-32768 bis 32767 oder -2147483648 bis 2147483647	`5 -10 0xff 084`
signed int	2 oder 4	-32768 bis 32767 oder -2147483648 bis 2147483647	`5 -10 0xff 084`
unsigned int	2 oder 4	0 bis 65335 oder 0 bis 4294967295	`5U 0xffu 084U`
long	4	-2147483648 bis 2147483647	`5l -10l 0xffL 084L`
signed long	4	-2147483648 bis 2147483647	`5l -10L 0xffl 084l`
unsigned long	4	0 bis 4294967295	`5uL 0xffUL 084Ul`
float	4	3.4E +/- 38 (7 Ziffern nach Komma)	`3.5f -12.34f 2e-3f`
double	8	1.7E +/- 308 (15 Ziffern nach Komma)	`3.5 -12.34 2e-3`
long double	10	1.2E +/- 4932	`3.5L -12.34L 2e-3L`

Wie du siehst, gibt es auch zu Ganzzahltypen kleine Anhängsel. **long** sollte mit `l` oder `L` und **unsigned int** oder **unsigned short** sollten mit `u` enden. Diese Buchstaben verdeutlichen dem Compiler ausdrücklich, dass die Konstanten von einem bestimmten Typ sind. Du kannst die Buchstaben groß oder klein schreiben, es spielt im Gegensatz zum Rest von C++ keine Rolle. Ich empfehle, weil das kleine '`l`' fast aussieht wie die eins '`1`', die Buchstaben groß zu schreiben.

Type-cast

long, **int** und **short** sind Ganzzahlentypen. **float** und **double** sind Fließkommazahlentypen. **char** ist eigentlich auch eine Ganzzahl. Wenn du einen **float**- oder **double**-wert einem **long**, **int** oder **short**-Wert zuweisen würdest, würde der Compiler eine Warnung ausgeben:

```
int A1 = 10;
float B1 = 20.55f;
char C1 = 100;
short D1 = 128;

A1 = B1; //A1 = 21; der Compiler rundet 20.55 auf und gibt eine
Warnung aus
C1 = A1; // das geht!!
A1 = D1; // das geht auch!!
```

Im folgenden will ich dir 3 Typumwandlungsoperatoren vorstellen, mit denen du dem Compiler klarmachen kannst, dass du die Umwandlung beabsichtigst. Die ersten zwei sind eher für einen alten Stil der Programmierung, ich will sie dir nur zeigen, damit du von ihnen mal gehört hast.

1.) Traditioneller C-Cast

Dieser Castingoperator stammt vom ursprünglichen C. Dabei kommt der Typ, der erwünscht wird für einen Ausdruck, in Klammern:

```
(Typ) Ausdruck
```

Ein kleines Beispiel:

```
int A1 = 10;
float B1 = 20.55f;
A1 = (int)B1; // keine Compiler-Warnung
```

2.) Funktionaler Cast

Dieser ist eigentlich nur eine Abwandlung des traditionellen C-Casts. Hier ist es genau anderherum: der Ausdruck steht in Klammern und der gewünschte Typ steht davor:

```
Typ (Ausdruck)
```

Als Beispiel:

```
int A1 = 10;
float B1 = 20.55f;
A1 = int(B1); // keine Compiler-Warnung
```

Seinen Namen hat dieser Cast davon, dass er einem Funktionsaufruf gleicht. Er ist nur für die bereits vorgestellten Datentypen verfügbar.

3.) Der C++-Cast **static_cast**<>()

Dieser ist einer der 4 C++-Casts. Er soll die beiden obigen alten Casts ablösen, denn er tut eigentlich dasselbe. Allerdings gelten die neuen Casts als sicherer und auch als besser einsetzbar, wenn auch schreibaufwendiger.

static_cast<>() soll uns also dazu dienen, einen normalen Cast durchzuführen. Er hat folgenden Syntax:

```
static_cast<Typ>(Ausdruck)
```

Wieder ein kleines Beispiel:

```
int A1 = 10;
float B1 = 20.55f;
A1 = static_cast<int>(B1); // keine Compiler-Warnung
```

Beispiel cast1.

Wenn also eine Typumwandlung geschehen soll, nimmst du natürlich den neuen glänzenden C++ Cast!

Typedef

Du kannst auch neue Typen definieren, diese basieren jedoch auf schon bekannten Typen. Das geht mit dem Schlüsselwort **typedef**:

typedef BekannterTyp NeuerTyp;
BekannterTyp: Hier muss entweder ein bekannter Typ (**float**, **int**, **char** ...) oder ein bereits von dir mit **typedef** definierter Typ hin. NeuerTyp: Hier muss der Name für den neuen Typen hin.

Beispiel typedef1.

Zusammenfassung

Dieses Kapitel hat dir wieder ein Stückchen C++-Weisheit gebracht. Du hast dein Wissen über Variablen weiter ausgebaut. Hoffentlich hast du alles gut nachvollziehen können, denn Variablen und Konstanten sind wirklich unerläßlich beim Programmieren.

Ganz am Anfang hab ich dir die Konstanten erklärt. Es gibt zwei Unterarten - literale und symbolische Konstanten. Zu den symbolischen gibt es wiederum 2 Varianten: erstens mit #define und zweitens mit **const**. Dann hast du etwas über Benennungsregeln und Gültigkeitsbereiche von Variablen und Konstanten erfahren. Und zu guter letzt hast du dich mit Variablenzusätzen, dem Type-Casting und dem Definieren neuer Typen rumgeschlagen. (Jetzt gerade liest du diesen Satz hier und der endet auch gleich ;)

Alle hier vorgestellten Datentypen sind einfache Datentypen - es gibt neben denen noch komplexe Datenstrukturen und Typen.

Workshop

Zeit für eine Kontrolle

1.) Welche einfachen Datentypen gibt es in C++??
2.) Wie deklariert man eine Variable und eine Konstante??
3.) Wie kann man einen Typ für **unsigned char** definieren??
4.) Was bedeutet denn Type-Cast??
5.) Was ergeben die folgenden Ausdrücke??

Definitionen: **char** A = 'B'; **const int** B = 23; **float** C = 0.7f;				
A / C	A * 1.5f	A * (255-A)	A / 3	A = B + C;
0x123 + 012	0xF * 0xF	035 / C	0x35 + 'f'	B += A;
C / 2	C / 2.0f	C * B = A;	A = B * 023	C = A / C;

Einige davon kannst du im Kopf rechnen. Um jedoch das genaue Verhalten des Compilers herauszufinden, solltest du ein Programm dazu erstellen.

6.) Finde passende Datentypen und geeignete Namen für:

- **typedef unsigned char**
- **typedef unsigned int**
- Preis für ein Buch
- Fläche von ganz Deutschland in km^2
- Masse der Sonne

7.) Welche Unterschiede gibt es zwischen mit #define und mit **const**

definierten Konstanten??

Programme

1.) Schreibe ein Programm, das deinen Namen mit mehreren **char**-Konstanten und –Variablen ausgibt
2.) Schreibe ein Programm, das deinen Namen als Zeichenkettenkonstante mit `cout` ausgibt, benutze dabei `#define`, um den Ausdruck mit einem Buchstaben aufrufen zu können

Link:

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop3_3.html

Kapitel 4

Bis jetzt waren alle Programme simple Folgen aus Befehlen, langweilig und monoton. Jetzt kommt etwas Abwechselung rein, hier lernst du:

- einfache Fallunterscheidung mit **if**
- verzweigte Fallunterscheidung mit **else** und **else if**
- Mini-Fallunterscheidung mit dem ?:-Operator
- Fallunterscheidung mit **switch**

Fallunterscheidung mittels if

Mit dem Schlüsselwort **if** kannst du flexibel Einzelfälle unterscheiden lassen. Dabei sagst du dem Compiler, unter welcher Bedingung der gewünschte Programmteil ausgeführt werden soll. Der Syntax von **if** ist:

```
if( Bedingung )
{
   //Anweisungen
}
```

WENN die Bedingung wahr ist, DANN werden die Anweisungen in den geschweiften Klammern ausgeführt. Um eine solche Bedingung zu schildern, gibt es die logischen und die vergleichenden Operatoren. Im Kapitel 2 hab ich Variablentypen erklärt - die Booleans kommen hier bei der Fallunterscheidung sinnvoll zum Einsatz.

bool ist ein Typ, der nur 2 Werte speichern kann – 0 (**false**) und 1(**true**). Genauer gesagt kann dieser 1-Byte-Typ genauso viele Werte speichern wie ein **char**. Es geht aber eigentlich nur um 0(**false**) und nicht-0(**true**).

Vergleichsoperatoren

Mit ihnen kann man prüfen, ob und auf welche Weise zwei Werte sich unterscheiden, dabei wird ein Wahrheitswert zurückgegeben, also ein **bool**. Folgende Tabelle zeigt diese Operatoren:

Operator	Name	Erklärung	Beispiel mit **int** A = 4;
==	gleich	Wahr, wenn linker Wert gleich dem rechtem ist	A == 5 wird zu **false**
!=	ungleich	Wahr, wenn linker Wert sich vom rechtem Wert unterscheidet	A != 5 wird zu **true**
<	kleiner als	Wahr, wenn linker Wert kleiner als der rechte ist	A < 4 wird zu **false**
<=	kleiner gleich	Wahr, wenn linker Wert kleiner oder gleich dem rechtem ist	A <= 4 wird zu **true**
>	größer als	Wahr, wenn linker Wert größer als der rechte ist	A > 4 wird zu **false**
>=	größer gleich	Wahr, wenn linker Wert größer oder gleich dem rechtem ist	A >= 4 wird zu **true**

In C++ gibt es mehrere Operatoren, die sich in ihren Symbolen ziemlich ähnlich sehen, so ist z.B. der Zuweisungsoperator (=) und der Vergleichsoperator (==) für viele Anfänger eine fatale Fehlerquelle. Also unbedingt ein bisschen üben, bis sich das eingefleischt hat!!

Logische Operatoren

Hier gibt es nur drei Operatoren:

Operator	Name	Allg. Beispiel	Erklärung	Beispiel mit `int A = 4` und `int B = 6`
&&	UND	`Ausdruck1 && Ausdruck2`	Wahr, wenn `Ausdruck1` und `Ausdruck2` zutreffen	`(A == 4) && (B == A + 2)` wird zu **`true`**
||	ODER	`Ausdruck1 || Ausdruck2`	Wahr, wenn `Ausdruck1` oder `Ausdruck2` zutreffen	`(A == 6) || (B == A - 2)` wird zu **`false`**
!	NICHT	`!Ausdruck`	Wahr, wenn `Ausdruck` nicht zutrifft	`!(B == A)` wird zu **`true`**

Ein bisschen praktisch angewandt:

```
bool a = false;
bool b = true;
short c = 0;

a = !c;     // a = nicht 0 , also a = 1
b = c && a; // b = 0 , weil a = 1 aber c = 0
c = 200;
a = (c==200) || (b!=1); // beide Bedingungen treffen zu, und
deswegen ist a = 1
```

Beispiel bool1.

Wie du siehst, kann man auch mehr als zwei Werte vergleichen.

Ein Anfängerfehler

Zunächst könnte es dir Probleme bereiten, wenn du aus der Formulierung "wenn x 4 oder 9 ist" die falschen Bedingungen ableitest:

```
if(x == 4 || 9)
{
   //Anweisungen   //werden IMMER ausgeführt!!
}
```

x wird lediglich daraufhin geprüft, ob es den Wert 4 hat. 9 wird überhaupt nicht mit x verglichen, und da 9 nicht 0 ist, nimmt das Programm **`true`** an. Also immer schön Acht geben!

Doch nun wieder zu Fallunterscheidung mit if:

Beispiel if1.

Hier kam die **`if`**-Abfrage zum Einsatz. Wenn du ein Programm schreiben würdest, das eine **`short`**-Variable nach seinem Wert abfragt, müsstest du jedoch alle möglichen Fälle abfragen (also 65536 **`if`**-Abfragen), wenn du die Vergleiche mit == vornimmst. Wenn du hier jedoch nur 2 exakte Werte brauchst (**`if`**`(a == 0)` und **`if`**`(a == 1)`, jedoch kein größer oder kleiner), kannst du restlich Fälle mit **`else`** und **`else if`** abfragen:

```
short a;
if(a==0)
{
   Anweisungen;
}
else if(a==1) // falls die vorige if-Abfrage nicht zutraf, wird
diese Bedingung geprüft
{
   Anweisungen;
}
else // falls keine der vorigen Abfragen zutraf, werden diese
```

```
Anweisungen ausgeführt
{
   Anweisungen;
}
```

Wenn `a==0` ist, dann werden die Anweisungen in der ersten **`if`**-Abfrage ausgeführt, die anderen jedoch nicht. Falls `a==1` ist wird die erste **`if`**-Abfrage übersprungen und die Anweisungen in der **`else if`**–abfrage ausgeführt. Wenn keiner der beiden Fälle zutrifft, werden die Anweisungen in dem **`else`**-Zweig ausgeführt. Die **`else`** und **`else if`**-Erweiterungen beziehen sich immer auf das letzte **`if`**, das am DIREKT davor stand. **`else if`**-Erweiterungen werden nur dann geprüft, wenn die letzte **`if`**- oder **`else if`**-Abfrage nicht zutraf. Die **`else`**-Erweiterung stellt den Schluss einer jeden **`if`**- bzw. **`else if`**-Abfrage dar, und kann nicht zweimal direkt hintereinander stehen.

Beispiel if2 zeigt einen solchen Zweig.

Wie du siehst, kann man auch mehrere **`if`**s ineinander verschachteln. Die inneren **`if`**s sind natürlich vollkommen unabhängig von den äußeren.

Kurze if-Anweisungen

Es ist auch möglich, EINE Anweisung direkt hinter den Vergleich zu schreiben (anstatt einen ganzen Block zu schreiben):

```
if(Variable == 22) cout << "Text";
```

Dann dürfen allerdings keine weiteren Anweisungen folgen, sonst würden diese auf jeden Fall ausgeführt werden, und nicht von der **`if`**-Abfrage betroffen sein.

Beachte dies, denn es ist noch so eine anfängliche Fehlerquelle, gleich nach der Bedingung das Semikolon zu setzen:

```
if(Variable == 22); cout << "Text"; //passiert eher selten

if(Variable == 11); //passiert deutlich öfter
{
   cout << "Variable";
   Variable++;
}
```

Beidesmal würden die Anweisungen ausgeführt werden, unabhängig davon, was `Variable` für einen Wert hat.

Der ?: -Operator - ein Vereinfachtes if-else

Ein weiterer Operator ist der ternäre Operator `?:`. Ternär heißt, dass er gleich 3 Operanden verlangt. Er prüft erst eine Bedingung und liefert bei wahrer Bedingung den einen Wert zurück, bei falscher Bedingung den anderen. Meist wird das angewandt, um einer Variablen einen Wert zuzuweisen:

```
int a = (b <= 5) ? b : 6;
```

`a` wird auf jeden Fall ein Wert kleiner oder gleich `6` zugewiesen. Nur wenn `b` größer als `5` ist, wird `a` zu `6`.

Zur besseren Lesbarkeit solltest du ausreichend Leerzeichen einsetzen. Wie gesagt findet dieser Operator fast ausschließlich Anwendung bei Zuweisungen. Extra zu diesem Zweck sind Makros programmiert wurden, die später behandelt werden!

Dieser Operator mag vielleicht etwas seltsam aussehen, stellt aber eine ernsthafte Alternative zum **if**-**else**-Geäst dar, wenn sie auch schlechter lesbar sind. Hast du schwierige Bedingungen, solltest du dann doch besser **if**-**else** nehmen!

Fallunterscheidung mit switch

Wenn du sehr viele Werte auf Gleichheit überprüfen willst, empfiehlt sich nicht, das mit **if** zu tun, denn es gibt noch einen weiteren Weg:

```
switch(Variable)
{
   case 'A' : Anweisungen1;
              break;
   case 'B' : Anweisungen2;
   case 67  : Anweisungen3;
              break;
   default  : Anweisungen4;
              break;
}
```

Hier wird keine Bedingung zur Überprüfung herangezogen, sondern eine Variable. Die zu überprüfende Variable steht dabei in den Klammern hinter dem Schlüsselwort **switch**. In den geschweiften Klammern wird die Variable mit möglichen Werten verglichen (auf Gleichheit, nicht kleiner oder größer als), wobei es sich hier um eine **char**-Variable handelt. Hinter das Schlüsselwort **case** kommt der mögliche Wert, als Buchstabe in einfachen Anführungszeichen oder als Zahl ohne Anführungszeichen.

Danach folgt ein Doppelpunkt, ab hier gehen die Anweisungen für den jeweiligen Fall los. Die Anweisungen werden bis zu dem nächsten Schlüsselwort **break** ausgeführt. Das heißt, dass bei **case** `A` die `Anweisungen1` ausgeführt werden, bei **case** `B` jedoch werden die `Anweisungen2` und `Anweisungen3` ausgeführt, weil zwischen den beiden Fällen `B` und `67` (also '`C`') kein **break** steht. Das Schlüsselwort **default** übernimmt alle Fälle, die nicht von den **case**-Abfragen abgefragt werden (praktisch wie das **else** von der **if**-Fallunterscheidung). Auch hier musst du den Doppelpunkt und **break** hinschreiben; außerdem solltest du die **default**-Anweisung IMMER einsetzen.

Beispiel switch1.

Zusammenfassung

Der Inhalt dieses Kapitels bezog sich sowohl auf Bedingungen (und deren Operatoren) als auch auf selektive Programmierung. Ich habe dir sämtliche Operatoren gezeigt, um Bedingungen in C++ zu formulieren und sie in Fallunterscheidungsanweisungen einzubringen.

Ein **if**-Statement besteht aus einer Bedingung und den Anweisungen, die der Computer bei erfüllter Bedingung ausführen soll. Die **else**-**if**-Anweisung kannst du dann einsetzen, wenn ein **if** missglückte, aber eine andere Bedingung vielleicht wahr sein könnte. Das **else** ist der krönende Abschluss - wenn kein vorhergehendes **if** oder keine vorhergehenden **else**-**if**s in Erfüllung gingen, werden die Anweisungen im **else**-Block auf jeden Fall ausgeführt. **else**-**if** und **else** sind optional.

switch dient lediglich dazu, viele **if**-Abfragen auf Gleichheit einer Variable zu vereinfachen. Das Schlüsselwort **break** dient zum Verlassen des **switch**-Blockes. Wird ein **case** einmal bearbeitet und endet nicht mit einem **break**, so wird ein danach folgendes **case** auch ausgeführt, eben bis ein **break** kommt.

Mit Bedingungen geht es im folgenden Kapitel weiter. Doch nun ist es wieder Zeit, um dumme Fragen zu beantworten...

Workshop

Fragen

1.) Was ergeben folgende Ausdrücke insgesamt - **true** oder **false**?? Definition:
```
int a = 5; int b = 0;
```

`-(-(-(-1)))`	`true + false`	`0`
`1`	`-1`	`a == 5`
`a = b`	`a * b`	`1 = a`

2.) Wozu gibt es `?:` ??
3.) Wie würdest du das hier in C++ schreiben??

1.)	Wenn x 10 ist und y 13, dann ...
2.)	Wenn x oder y 13 ist, dann ...
3.)	Wenn x 1 oder 5 und y 10 oder 20 ist bzw. wenn x größer als 50 ist, dann ...

4.) Versuche für folgende Bedingungen eine **if**-**else**- und eine **switch**-Implementation:

Wenn x 100, 200 oder 300 ist, dann vervierfache x!
Wenn x zwischen 100 und 110 liegt, dann lasse x zu 400 werden!
Wenn x negativ ist, mache es positiv!
Wenn nichts von dem zutrifft, dann lasse x zu 0 werden!

Programme

1.) Wenn du es noch nicht getan hast, entwickle ein Programm zu den Implementationen aus der vierten Frage, das den Benutzer x eingeben lässt.
2.) Schreibe ein Programm, das drei Fragen stellt, und zu jeder Frage Drei Antworten ermöglicht, also praktisch ein Quiz.

Links:

http://www.volkard.de/vcppkold/if_und_else.html **http://www.volkard.de/vcppkold/switch.html**
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop5_3.html
http://www.schornboeck.net/ckurs/bedingungen.htm

Kapitel 5

Wie du sicherlich schon gemerkt hast sind mehrmals hintereinander ausgeführte Befehle oder Befehlsgruppen eine unschöne Tipparbeit. Moderne Programmiersprachen liefern einen recht einfachen Ansatz - die Schleifen. Mehrmaliges Ausführen der selben Befehle mit Schleifen nennt man Iteration. (Die Alternative Möglichkeit ist Rekursion, die du später noch lernen wirst)

Schleifen

Mit Hilfe von Schleifen kannst du einen Teil des Programms beliebig oft wiederholen lassen. Das wird jeweils solange gemacht, bis eine Bedingung nicht mehr zutrifft. Allgemein gesehen unterscheidet man zwischen 3 Schleifen in C++, die ich dir nun nacheinander vorstellen werde.

Die while-Schleife

Der Syntax dieser Schleife:

```
while(Bedingung)
{
   //Anweisungen
}
```

Zuerst wird die Bedingung geprüft; wenn sie wahr ist, dann werden die Anweisungen ausgeführt, andernfalls nicht. Die Bedingung kannst du wie im vorigen Kapitel mit Vergleichsoperatoren und mit logischen Operatoren ausgedrücken.

Stimmt nach einer Durchführung die Bedingung immer noch, so geht es in die zweite Runde. Das ganze wird solange getan, bis die Bedingung nicht mehr stimmt.

Im folgenden Programm würde solange ein Ausrufezeichen ausgegeben, bis 'x' eingegeben wird:

```
int main(void)
{
   char i = 0;

   while(i != 'x')
   {
      cout << '!';
      cin >> i;
   }

   return 0;
}
```

Das folgende Projekt gibt genau 1000 Ausrufezeichen aus:

Beispiel while1.

Eine potentielle Fehlerquelle (wie beim **if**) könnte sein, gleich nach der Bedingung ein Semikolon zu schreiben:

```
while(Bedingung);
{
   //Anweisungen
}
```

Hier könnte leicht eine Endlosschleife entstehen. Bei einer Endlosschleife ist es problematisch, wieder herrauszukommen - als einzige Möglichkeit bleibt, das Programm zu beenden.

Die do-while-Schleife

Diese ist der **while**-Schleife sehr ähnlich, mit dem Unterschied, dass die Bedingung erst nach einem Durchlauf der Anweisungen überprüft wird. Ist diese wahr, so werden die Anweisungen solange ausgeführt, bis die Bedingung nicht mehr wahr ist.

```
do
{
   //Anweisungen
}while(Bedingung);
```

Hier muss der Bedingung ein Semikolon folgen. Fehlerquelle könnte sein, dieses zu vergessen oder eines nach dem **do** zu schreiben.

Mit dieser Schleife kannst du sicher sein, dass die Anweisungen mindestens einmal ausgeführt werden. So wird hier einmal '!' ausgegeben, weitere nach Bedarf:

```
int main(void)
{
   char i = 65;

   do
   {
      cout << '!';
      cin >> i;
   }while(i != 'x');

   return 0;
}
```

Beispiel dowhile1.

Die for-Schleife

Der Schleifenkopf (wo die Bedingungen drinstehen) ist ganz anders als bei den anderen beiden:

```
for(int i = 0; i <= 10; i++)
{
   //Anweisungen
}
```

Der Rumpf (wo die Anweisungen stehen) ist wie bei der **while** und **do-while**-Schleife. Der Schleifenkopf besteht allerdings aus mehreren Ausdrücken:

```
for( Initialisierung ; Bedingung ; Anweisung )
```

Der **for**-Schleife wird fast immer eine neue Variable, die Zählervariable, zur Verfügung gestellt. Hier können alle einfachen Datentypen (egal ob Ganz- oder Fließkommazahl, aber auch **bool** und **char**) eingesetzt werden. Die Werte sollten dabei gleich mit initialisiert werden (bzw. zugewiesen, wenn die Variable auch schon außerhalb der Schleife verfügbar gewesen ist). Die Initialisierung kann allerdings auch ganz weggelassen werden, dann muss halt schon eine Variable vorhanden sein, um als Zählervariable eingesetzt werden zu können. Meistens wird ein `int i` als Zählervariable eingesetzt.

Die Bedingung wird, wie zuvor auch schon, mit Vergleichsoperatoren und mit logischen Operatoren ausgedrückt. Bei der Anweisung kannst du einen geeigneten Ausdruck hinschreiben, etwa `i++` , der die Zählervariable verändert. Allerdings kann auch die Anweisung weggelassen werden, dann sollte diese Zähler-Anweisung aber im Rumpf stehen.

Die **for**-Schleife arbeitet nach folgendem Schema:

1.) Variable initialisieren
2.) Bedingung prüfen und entweder
  - in den Rumpf einsteigen oder
  - Schleife verlassen
3.) Anweisungen im Schleifenkopf ausführen
4.) Wiederholung ab 2.)

Bis auf den ersten Durchlauf werden die Anweisungen im Schleifenkopf vor dem Prüfen der Bedingung ausgeführt!

Beispiel **for1** macht genau dasselbe wie die Beispielprogramme mit den **while**- und **do-while**-Schleifen, nur unter Verwendung einer **for**-Schleife.

Sprunganweisungen

Sprunganweisungen dienen dazu, im Programm zu verschiedenen Punkten zu springen. Dabei werden die Werte in den Variablen beibehalten, höchstens werden Variablen gelöscht, deren Gültigkeitsbereich verlassen wird. Eine sehr bekannte, wenn auch verpöhnte Sprunganweisung ist das **goto**.

break in Schleifen

Das Schlüsselwort **break** hat auch bei den Schleifen eine Bedeutung: du kannst durch dieses Wort eine Schleife verlassen, auch mittendrin oder am Anfang:

```
int a;

for(int i = 0; i < 10; i++)
{
   cout << "Gib bitte eine Zahl ein : ";
   cin >> a;
   if(a==0) break; //Schleife wird nur einmal durchlaufen
}
```

Die **for**-Schleife ist durch die Initialisierung einer Variable besonders vorteilhaft, da du die zugehörige Zählervariable meistens sofort im Schleifenkopf findest. Wenn du einen solchen Schleifenkopf verwendest, ist es auch kein großes Problem, aus Versehen ein Semikolon danach zu setzen.

Wenn es jemals nötig ist, eine Endlosschleife zu Programmieren, so musst du sie mit break verlassen. Tatsächlich gibt es nur diese Lösung neben der, das Programm zu beenden. Im realen DOS wäre das allerdings problematisch, denn du müsstest den Computer neustarten. Ein Glück, dass es die Eingabeaufforderung gibt.

continue

Mit **continue** kannst du eine Schleife dazu veranlassen, einen Durchgang, der gerade ausgeführt wird, abzubrechen und gleich mit dem nächsten zu starten. Hier ein Beispiel:

```
int a;

for(int i = 0; i < 10; i++)
{
   a = i;
   if(a == 2) continue; //Wenn a == 2, werden nachfolgende
Anweisungen nicht ausgeführt
                        //und der nächste Durchgang gleich
gestartet
   cout << a << '\n';
}
```

goto

Es gibt noch eine Möglichkeit, Schleifen zu erzeugen:

```
int a = 0;
START:
cout << a << '\n';
a++
if(a < 10) goto START;
```

Mit `START:` wird der Startpunkt definiert (wird auch "Label" genannt). Solange `a` kleiner als `10` ist, veranlasst der Befehl **`goto`** `START;` das Programm, bei `START` erneut loszulegen. Das funktioniert aber nur innerhalb einer Funktion, nicht Funktionsübergreifend.

Ein **goto** anzuwenden gehört aber nicht zu einem guten Programmierstil. Erstens lässt sich ein damit erreichtes Ziel meist auch ohne das **goto** erreichen, und zweitens ergibt sich bei mehreren **goto**s ein schwer lesbarer

Code (schon die Labels verwirren) - manchmal auch Spaghetti-Code genannt. Daher ist ein **goto** nur im äußersten Notfall anzuwenden!!

Beispiel für **break**, **continue** und **goto**: **goto1**.

Zusammenfassung

Mit Schleifen kannst du deine Programme sehr gut strukturieren. Man hat für verschiedene Einsatzbereiche drei Schleifenarten erfunden: die **while**-, **do**-**while**- und **for**-schleifen. Die **for**-Schleife bringt meist eine Zählervariable mit, um die Anzahl der Durchgänge einfach festzuhalten.

Anweisungen zum Springen im Code sind manchmal ganz nützlich, kommen aber in der Regel sehr selten vor. Auf keinen Fall solltest du **goto** verwenden, denn der Code ist bei umfangreichen Projekten schwer lesbar und anpassbar.

Workshop

Fragen

1.) Was sind die Unterschiede zwischen den einzelnen Schleifen??
2.) Kann man mehr als eine Zählervariable in einer **for**-Schleife initialisieren bzw. verwenden??
3.) Kann man eine **for**-Schleife in eine **while**-Schleife reinschreiben??
4.) Gibt es einen Befehl, mit dem man mehrere Durchgänge einer Schleife auf einmal überspringen kann??
5.) Was ist der Unterschied zwischen iterativer und rekursiver Wiederholung??
6.) Was ist der Unterschied zwischen `for(;i < 10;)` und `while(i<10)`??
7.) Was sind Endlosschleifen??

Programme

1.) Schreibe ein Programm, das drei Fragen stellt, und zu jeder Frage Drei Antworten ermöglicht, also praktisch ein Quiz.
2.) Schreibe ein Programm, das die Zahlen 1 bis 10 in jeweils einer neuen Zeile ausgibt, benutze dabei alle drei Schleifen.

Weitere Programme musst du dir selber ausdenken, du kannst etwa primitive Textadventures oder Rundenkämpfe programmieren. Falls du bisher nur Programme mit maximal 25 Zeilen im Quellcode geschrieben hast, solltest du mal testen, ob du auch schon größere Programme hinkriegst. Auch einfache Textadventures können dir dabei helfen, deinen eigenen Programmierstil zu entwickeln. Außerdem sammelst du Erfahrung mit größeren Projekten, die dir später mal sehr zunutze sein kann.

Wohlwissend, dass sowas ein bisschen Zeit (bis 2 Wochen) beansprucht, kann ich dir das nur wärmstens empfehlen. Übung macht den Meister und was klein Hänschen nicht lernt, das lernt Hans nimmermehr :o

Links:

http://www.volkard.de/vcppkold/whileschleife.html http://www.volkard.de/vcppkold/forschleife.html http://www.volkard.de/vcppkold/doschleife.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop5_4.html
http://www.schornboeck.net/ckurs/schleifen.htm

Kapitel 6

Bereits jetzt hast du Millionen von Möglichkeiten, eigene Programme zu gestalten. Nachdem du genug geübt hast, soll es weitergehen mit:

Funktionen

Bisher liefen alle Befehle in der `main`-Funktion ab, wie etwa Ausgaben mit `cout`, Zuweisungen oder Operationen. Mit Funktionen kann man Teile der `main`-Funktion oder sogar den ganzen Inhalt nach außen verlagern. Es handelt sich also um kleine Unterprogramme im Hauptprogramm, das aus der `main`-Funktion besteht. Funktionen dienen dazu, um Code, der mehrmals in Gebrauch ist, allgemein zur Verfügung zu stellen.

Um Funktionen zu erstellen, kannst du zwei Wege gehen:

1.)
- Funktionsprototyp vor der `main`-Funktion deklarieren
- Funktionsinhalt nach der `main`-Funktion definieren

oder

2.) - ganze Funktion vor der `main`-Funktion definieren

Prototypen deklarieren

Ein Prototyp ist eigentlich nur ein Funktionskopf, der dazu da ist, um die Funktion erst einmal bekannt zu geben (deshalb heißt es Prototyp deklarieren). Jede Funktion kann einen Wert zurückgeben und Parameter verlangen. Dies wird neben dem Funktionsnamen bekannt gegeben.

Die allgemeine Form für einen solchen Prototypen sieht so aus:

```
RückgabeTyp FunktionsName (Parameter);
```

Diese Zeile kommt noch vor die `main`-Funktion:

```
int Funktion(int i); // Funktionsdeklaration oder Prototyp

int main(void)       // Hauptfunktion
{
   //Anweisungen...
}
```

Wie du siehst, können bei dem Rückgabetyp und den Parametern einfache Datentypen eingesetzt werden, es funktionieren aber auch selbsterstellte bzw. komplexe Datentypen (die später behandelt werden). Hier haben wir die Funktion "`Funktion`" mit dem Rückgabetyp **`int`** und dem Parameter `i` vom Typ **`int`**.

Was bedeutet dieses void bei der main-Funktion?

In der Parameterliste kommt ständig dieses **void** vor - es kam nicht bei den einfachen Datentypen vor und dieser Parameter hat auch keinen Namen. Welche Bedeutung hat es??

void repräsentiert das Nichts. Eine Funktion kann einen Wert zurückgeben und Parameter verlangen, muss dies aber nicht. Wenn so etwas vorkommt, setzt du eben das **void** ein. Die main-Funktion verlangt also keinen Parameter.

Man kann es auch weglassen, was dann aber kein guter Programmierstil ist. Bisher haben wir nur mit **int** main(**void**) gewerkelt. Schon bald wirst du aber noch mit einer anderen Form umgehen können.

Funktionen definieren

Nun, die Schnittstelle - der Funktionskopf - ist bekannt. Es fehlt uns allerdings noch an Inhalt. Der Funktionsrumpf kommt bei der Definition der Funktion nach dem Funktionskopf:

```
RückgabeTyp FunktionsName (Parameter)
{
   //Anweisungen
}
```

Die Definition stellst du dann nach die main-Funktion:

```
int Funktion(int i); // Funktionsdeklaration

int main(void)       // Hauptfunktion
{
   //Anweisungen
}

int Funktion(int i) //Funktionskopf
{                   //Funktionsrumpf Anfang
   int z = i * 100;
   return z;
}                   //Funktionsrumpf Ende
```

In der ersten Zeile wird die Funktion deklariert. Im letzten Abschnitt wird die Funktion definiert, d.h. sie wird mit Inhalt belegt.

Variante Zwo

Die zweite Variante, um Funktionen zu erstellen, ist es, die komplette Funktionsdefinition vor die main-Funktion zu schreiben. Damit wird ein Prototyp überflüssig.

Was an der ersten Variante besser ist?? Mit Prototypen gibt man dem Compiler die Funktionen erst einmal bekannt. Er weiß dann, dass es Funktion A gibt und wie diese Funktion aufgebaut ist. Punkt eins ist wichtig, wenn Funktion A eine Funktion B aufruft, die wiederum Funktion A aufruft. Der zweite Punkt ist, dass es einen Fehler hagelt, wenn Definition und Deklaration voneinander abweichen. Dadurch können andere Fehler vermieden werden, die später im Programm auftreten würden. Außerdem ist es übersichtlicher, wenn man gleich die main-Funktion zu Gesicht bekommt anstatt erst seitenweise scrollen zu müssen.

Aufrufen einer Funktion

Jetzt muss sie nur noch aufgerufen werden, damit sie etwas tut. Deklarierte Parameter musst du dabei mit Werten belegen - dabei kann es sich um literale oder symbolische Werte handeln (die eingesetzten Werte werden Argumente genannt). Im Beispiel `Funktion` müsste eine Ganzzahl an `i` übergeben werden:

```
int Funktion(int i);

int main(void)
{
   int u = 5;
   int a = Funktion(u);  //5 wird an i übergeben

   return 0;
}

int Funktion(int i)
{
   int z = i * 100;
   return z;
}
```

In der `main`-Funktion wird eine Variable `u` mit `5` deklariert, danach `a` mit dem Rückgabewert von `Funktion()` mit dem Argument `u`. Bei dieser Deklaration von `a` wird also `Funktion()` aufgerufen. In der Funktion selber hat der Parameter `i` den Wert von `u` erhalten. Nachdem `z` mit `i * 100` initialisiert wurde, gibt Funktion den in `z` gespeicherten Wert zurück.

Beispiel funktion1. Dort findest du die Prototypen der Funktionen `f1()` und `f2()`, die gleich nach der `main`-Funktion definiert werden. In der `main`-Funktion drin werden diese beiden insgesamt 6 mal mit jeweils unterschiedlichen Argumenten aufgerufen.

Als Überblick und Zusammenfassung:

Ein Prototyp ist die Deklaration einer Funktion. Es handelt sich dabei um den Funktionskopf.

Der Rückgabetyp gibt an, von welchem Typ die zu erwartende Rückgabe ist. Der zurückgegebene Wert wird an die Stelle eingesetzt, an der die Funktion aufgerufen wurde.

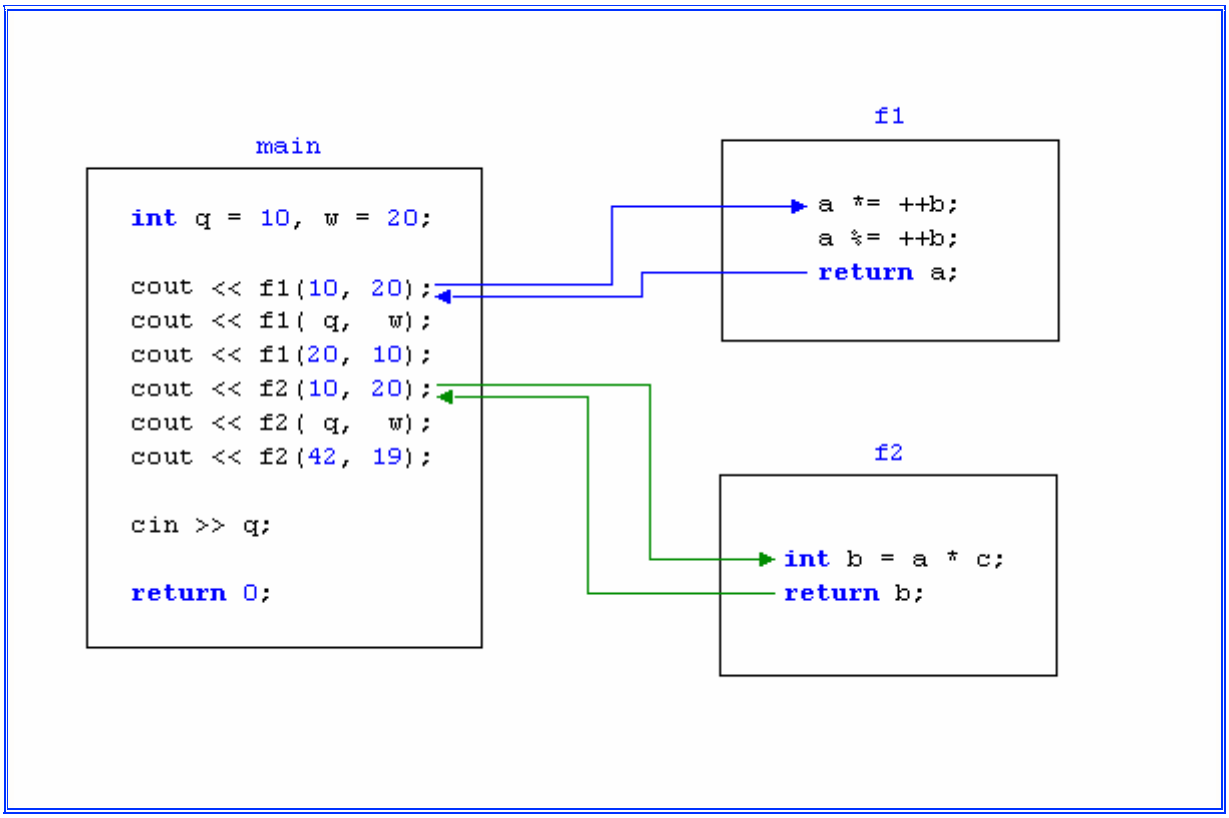
Der Funktionsname identifiziert die Funktion eindeutig. Er ist genau wie bei den Variablen frei wählbar, jedoch musst du dabei dieselben Regeln beachten wie bei den Variablen. Allerdings kann eine Funktion den selben Namen tragen wie eine Variable (der Compiler bemerkt den Unterschied wegen der Klammern).

Ein Parameter ist eine Variable (aus der Parameterliste), die in der Funktion lokal genutzt werden kann, also außerhalb nicht verfügbar ist.

Die übergebenen Variablen bzw. Konstanten (also die in die Parameterliste eingesetzten Werte) heißen Argumente.

Doch wie läuft das genau ab??

Am Projekt **funktion1** (von oben) kannst du sehr gut nachvollziehen, wie das Programm denn eigentlich abläuft:



Diese Darstellung ist realistisch - es werden tatsächlich Sprünge vollführt.

`main` - `f1()` - `main` - `f1()` - `main` - `f1()` - `main` - `f2()` - `main` - `f2()` - `main` - `f2()` - `main` - Ende

Gültigkeitsbereiche

Dabei werden jeweils die Variablen `a`, `b` und `c` mehrmals in den Funktionen benutzt. Diese sind für die Funktionen jeweils lokal und außerhalb nicht verfügbar. Außerdem wird in `f2()` die Variable `b` erstellt - diese ist lokal, also auch nur in `f2()` verfügbar. Auch die Variablen `q` und `w` sind in keiner der anderen Funktionen verfügbar.

Globale Variablen (richtig global, nicht nur in der aufrufenden Umgebung vorhandene) sind in der Funktion natürlich auch verfügbar. Bei gleichnamigen Variablen, die lokal und global vorkommen, musst du mit dem Gültigkeitsoperator `::` unterscheiden.

Parameter sind also lokale Variablen. Beim Funktionsaufruf `f1(q, w)` wird weder `q` noch `w` verändert, sondern immer nur die Parameter (Kopien von diesen Argumenten). Wenn die Argumente aus normalen Variablen bestehen, so nennt man einen Aufruf der Funktion "Call-by-Value". Später werden wir noch "Call-by-Reference" kennenlernen.

Beispiel funktion2.

In diesem Beispiel gibt es 4 mal die Variable `a`, jeweils mit unterschiedlichem Gültigkeitsbereich:

- `a` global
- `a` in der `main`-Funktion
- `a` in `f1()`
- `a` in `f2()`

Wichtig ist zu wissen, dass das globale `a` überall gilt, aber mit `::a` angesprochen werden muss, wenn es lokalere Variablen gibt.

Funktionen ineinander schachteln

Wie du schon gesehen hast, kannst du Funktionen von anderen Funktionen aus aufrufen. `f1()` und `f2()` wurden jeweils von der `main`-Funktion gestartet. `f1()` und `f2()` könnten sich eigentlich auch gegenseitig aufrufen:

```
int f1(int a);
int f2(int a);
int f3(int a);

int main(void)
{
   cout << f1(1);
   cout << f1(2);

   return 0;
}

int f1(int a)
{
   return a * f2(a);
}

int f2(int a)
{
   return a * a * f3(a);
}

int f3(int a)
{
   return a + 1;
}
```

Einen Aufruf der Funktion `f1()` kannst du dir dann so vorstellen:

```
int main(void)
{
   int a, f1_, f2_, f3_;

   { //f1()
      a = 1;

      { //f2()
         { //f3()
            f3_ = a + 1;
         }

         f2_ = a * a * f3_;
      }

      f1_ = a * f2_;
   }
   cout << f1_;

   { //f1()
      a = 2;

      { //f2()
```

```
            { //f3()
               f3_ = a + 1;
            }

            f2_ = a * a * f3_;
         }

         f1_ = a * f2_;
      }
      cout << f1_;

      return 0;
}
```

Hier ist es gleich ersichtlich: mit Funktionen kannst du den Code wesentlich besser strukturieren und kurz halten. Dieser große Block enthält alle wichtigen rechnerischen Anweisungen aus den Funktionen `f1()`, `f2()` und `f3()`. Mit einem einzigen Aufruf der Funktion `f1()` ist der Code im Kasten darüber aber sehr viel verständlicher. Die Blöcke könnten wir natürlich auch kürzen, aber es handelt sich bei Funktionen meist um allgemeine Lösungswege. Es könnte auch einmal ein Aufruf von `f2()` oder `f3()` allein vorkommen.

Funktionsaufrufe als Argumente

Wenn eine Funktion einen passenden Wert zurückgibt, ist es ohne weiteres möglich, eine Funktion als Argument für einen anderen Funktionsaufruf zu verwenden:

```
int f;

int f1(int a, int b);
int f2(int a);
int f3(int a);

int main(void)
{
   cout << f1(f2(5), f3(4));
   cout << f1(f2(6), f3(3));
   cout << f1(f2(7), f3(2));

   cin >> f;

   return 0;
}

int f1(int a, int b)
{
   return a + b;
}

int f2(int a)
{
   return a * a;
}

int f3(int a)
{
   return a + 1;
}
```

Was bei langen Parameterlisten wiederum zu unleserlichen Code führen kann.

Logisch ist, dass zuerst die Funktionen ausgeführt werden, die in einer Parameterliste stehen (bzw. die die innersten Aufrufe sind) und deren Rückgabewerte für andere Funktion als Argument dienen sollen. In der Parameterliste selbst gilt die Regel links vor rechts: erst wird `f2()` ausgeführt, danach `f3()`. Wenn die Funktionen mit denselben globalen Variablen arbeiten, ist dies umso wichtiger!

Das erklärt auch das Phänomen im Beispiel **funktion2** von oben. `cout` ist ein Objekt und es führt eine Funktion aus, wenn man schreibt `cout << "bla";`. Du musst dir die Auflistung von Strings und Variablen und Konstanten hinter dem `cout`-Objekt wie eine Parameterliste vorstellen:

```
cout << " - f1(10) ergibt: " << f1(10) << "\n";
```

übergibt `cout` drei Argumente - `f1(10)` beinhaltet als erste Anweisung `cout << a;`. Deswegen erscheint auf dem Bildschirm zuerst die 10 und dann " - f1(10) ergibt: ...".

Beispiel funktion3 verdeutlicht das nochmal.

Zusammenfassung

Und ein weiteres Kapitel ist geschafft! Funktionen erleichtern das Programmieren immens. Funktionen sind fast immer verallgemeinerte Vorgänge, die flexibel eingesetzt werden können. Nun kannst du deinen Code besser gliedern und klarer formulieren.

Eine Funktion muss entweder vor der `main`-Funktion definiert werden oder deklariert werden und nach der `main`-Funktion definiert werden. Der Einsatz von Prototypen ist für sauberen Code unerlässlich.

Jede Funktion kann sowohl Parameter verlangen als auch Werte zurückgeben. Diese können in einfachen und komplexen Datentypen auftreten. Mit **`void`** lässt sich das verhindern, einfach wenn keine Rückgabe oder kein Parameter nötig ist.

Die hier vorgestellten Funktionsaufrufe sind alle "Call-by-Value". Das heißt, dass die Argumente in die Parameter reinkopiert werden und somit die Argumente unangerührt bleiben.

Workshop

Fragen

1.) Welche Datentypen, die ich bisher erklärt hab, können Rückgabetyp und welche Parametertyp sein??
2.) Was ist, wenn kein Parameter verlangt werden soll??
3.) Wozu dienen Funktionen überhaupt??
4.) Was passiert, wenn du bei einer Funktion vergisst, den Rückgabetyp anzugeben??
5.) Wann solltest du globale Variablen einsetzen??
6.) Wo überall kann ein Funktionsaufruf stehen??
7.) Was gibt es zu beachten, wenn du eine Funktion aufrufst??
8.) Müssen Parameter immer nur Kopien der Argumente sein??
9.) Was für Fehler springen dir im folgenden Code sofort ins Auge??

```
#include <iostream>

using namespace std;

int f1(int a, b);

void main(void)
{
   int a = f1(23, 34);
   cout << "a : " << a;
   int b = f1(a, 100);
   cout << "b : " << b;

   cin >> a;

   return 0;
}

int f1(int a, b)
{
   return a+b*a-b+a*b*b-a*a*b+b/a;
}

int f2(int a, b)
{
   a = b;
}
```

Programme

1.) Schreibe ein Programm, das verschiedene Berechnungen und Ausgaben tätigt. Nutze die Macht (der Funktionen) zur Addition, Multiplikation, Substraktion und Division sowie zur Ausgabe!
2.) Schreibe ein Programm, das den Benutzer zwei mal zum Zahleneingeben bewegt, die Werte vergleicht und dann je nach Ergebnis einen Text ausgibt. Wieder Funktionen nutzen!

Links:

http://www.volkard.de/vcppkold/funktionen.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop7.html
http://www.schornboeck.net/ckurs/funktionen.htm

Kapitel 7

Dieses Kapitel ist als Erweiterung für den ersten Teil gedacht. Hier erkläre ich dir einige weitere wissenswerte Dinge betreffs Funktionen.

Rekursion

Im vorigen Kapitel hast du die Iteration kennengelernt. Rekursion ist die alternative Möglichkeit zum Wiederholen von Anweisungen.

Weiter oben hast du erfahren, dass Funktionen andere Funktionen aufrufen können. Eine rekursive Funktion geht einen Schritt weiter und ruft sich selber auf (direkte Rekursion) oder sie ruft eine andere Funktion auf, die dann wieder die erste Funktion aufrufen wird (indirekte Rekursion). Dies ist für einige Probleme eine gute Lösungsmöglichkeit - besonders im mathematischen Bereich.

Zur Rekursion gehört aber immer auch ein zweiter Teil. Ohne Abbruchbedingung gleicht eine rekursive Funktion einer Endlosschleife. Um das zu realisieren, muss jede Rekursionsfunktion eine Rückgabe machen und Parameter verlangen.

Es gibt zwei oft gebrachte Beispiele für die Rekursion: die Fakultät einer Zahl und die Fibonacci-Reihe. Beide funktionieren mit Ganzzahlen. Später wirst du eine Funktion zum Sortieren von Daten kennenlernen, und zwar der rekursive Quick Sort.

Fakultät

Die Fakultät einer Zahl ist die Zahl selbst mit all seinen Vorgängern bis zur 1 multipliziert. In der Mathematik drückt man z.B. die Fakultät von 5 mit 5! aus.

```
5! = 5·4·3·2·1 = 120
```

Verallgemeinert gilt:

```
n! = n·(n-1)·...·2·1
```

Wobei n größer oder gleich 2 ist. Die Fakultät von 0 sowie die Fakultät von 1 ist als 1 definiert.

Um nun zu einem gängigen Ausgangspunkt zu kommen:

```
n! = n·(n-1)!
```

So sieht unsere Implementation aus:

```
int Fakultaet(int n)
{
   if(n >= 0)
   {
      if(n == 0) return 1;
      else return n * Fakultaet(n-1);
   }
   else return 0;
}
```

Und tatsächlich gibt die zuerst aufgerufene Funktion letztenendes das komplette Produkt zurück. Dass `n` nicht kleiner als `0` sein darf, hat übrigens damit zu tun, dass `n` mit fortlaufenden Rekursionen immer negativer wird und die einzige Abbruchmöglichkeit darin besteht, den Wertebereich von **`int`** zu unterschreiten und so irgendwann zu `0` zurück zu gelangen.

Beispiel funktion4.

Fibonacci-Reihe

Diese Zahlenreihe besteht aus Zahlen, die sich aus den zwei vorhergehenden Zahlen bilden. Nach der zweiten Zahl der Reihe ist jede Zahl die Summe aus den beiden Zahen, die davor kamen. Der Anfang der Fibonacci-Reihe sieht so aus:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
```

Am Anfang steht die 1. Die zweite Zahl ergibt sich aus 0 + 1. Dann 1 + 1, 1 + 2 usw.

> Das Problem, das jetzt rekursiv zu lösen wäre: Welche Zahl kommt an 14ter Stelle??

Verallgemeinert könnte man das so Formulieren:

```
Fibonacci-Zahl n = (Fibonacci-Zahl n-1) + (Fibonacci-Zahl n-2)
```

Bedingung ist immer, dass n >= 3 ist. Für n == 2 oder 1 ist der Wert 1.

Um das rauszufinden, gibt es folgende Implementation:

```
int Fibonacci(int n)
{
   if(n < 3) return 1;
   else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Beispiel funktion5 versucht, die ersten 15 Fibonacci-Zahlen rauszufinden.

Inline-Funktionen

Mithilfe des Schlüsselwortes **`inline`** kannst du Funktionen so modifizieren, dass sie nicht nur einmal in den Speicher geschrieben, sondern immer da, wo sie benötigt werden. Wenn eine normale Funktion aufgerufen wird, springt das Programm zur Stelle, wo der Maschinencode der Funktion steht. Diese Sprünge können ein wenig Zeit in Anspruch nehmen. **`inline`**-Funktionen wollen da etwas verbessern - für kleine Funktionen, die oft ausgeführt werden, ist es sinnvoll, die Sprünge zu vermeiden und einfach den Maschinencode an Ort und Stelle des Aufrufs zu plazieren.

Das lohnt sich aber nur für kleine Funktionen, die den Code nicht allzu groß aufblähen. Große Funktionen, die selten aufgerufen werden, brauchen so etwas natürlich nicht. Hier ein Kollege, für den es sich lohnt, ihn **`inline`** zu machen:

```
inline void F1(void)
{
   cout << "haha";
}

int main(void)
{
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();

   return 0;
}
```

Hier wird mehrere male die Funktion `F1()` ausgeführt. Das fertige Programm hätte auch aus dem folgenden Code entstanden sein können:

```
inline void F1(void)
{
   cout << "haha";
}

int main(void)
{
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   //Und so weiter...

   return 0;
}
```

Diese Maßnahme kann - korrekt eingesetzt - die Geschwindigkeit durchaus erhöhen, aber nicht unbedingt dramatisch.

Standardparameter

Manchmal ist es so, dass einige Parameter bei mehreren Aufrufen gleich sind. Um das ein bisschen einfacher zu gestalten, kannst du Parametern Standardwerte verpassen, sodass du beim Aufruf nicht unbedingt alle Argumente reinschreiben musst:

```
void F1(int a, int b = 100)
{
   //Anweisungen;
}

int main(void)
{
   F1(44);
   F1(44, 99);

   return 0;
}
```

b ist, falls man keinen anderes Argument angibt, immer 100. Der zweite Aufruf in der main-Funktion hat also 99 anstatt 100 als Argument. Du kannst jedem Parameter einen Standardwert zuweisen, mit der Einschränkung, dass vor einem Parameter ohne Standardwert kein Parameter mit Standardwert in der Parameterliste stehen darf. Es müssen also immer erst die Parameter weiter rechts einen Standardparameter haben, bevor die Parameter weiter links einen bekommen.

Funktionen Überladen

Jetzt wird es mal wieder verwirrend: In C++ kannst du mehrere Funktionen gleich benennen, ihnen dabei jedoch unterschiedliche Parameter geben. Der Compiler kann selbst unterscheiden, welche Funktion ausgeführt werden soll:

```
void A(char A){}
void A(int A){}

int main(void)
{
   int a = 0;
   char b = 'a';
   A(a); // zweite Funktion, weil der Parameter ein int ist
   A(b); // erste Funktion, weil der Parameter ein char ist

   return 0;
}
```

Sowas nennt man Funktionen Überladen. Dabei zu beachten ist, dass zwar unterschiedliche Parameterlisten bei gleichen Rückgabetypen möglich sind, andersrum geht es aber nicht.

Bei der Übergabe kannst du natürlich auch den Type-cast verwenden, wenn es sich um unterschiedliche Datentypen handelt, und so verhindern, dass eine falsche Version der Funktion aufgerufen wird.

In einem späteren Kapitel wirst du mit Templatefunktionen eine etwas modernere C++-Variante des Überladens kennenlernen. Mach dich aber trotzdem mit dem jetzigen Thema vertraut, es könnte mal nützlich sein!!

Rückgabe und Parameter der main-Funktion

Schon im ersten Kapitel hab ich dir eine Form der `main`-Funktion vorgestellt:

```
int main(void)
{
   //...

   return 0;
}
```

Der aktuelle Standard schreibt vor, der `main`-Funktion einen Rückgabetypen zu geben. Der mit **`return`** zurückgegebene Wert sollte `NULL` sein, wenn das Programm richtig abgelaufen ist. Wenn ein Fehler im Programm aufgetreten ist, sollte `1` zurückgegeben werden.

Eine zweite Variante der `main`-Funktion hat 2 Parameter:

```
int main(int argc, char *argv[])
{
   //...

   return 0;
}
```

Diese Konstruktion erlaubt den Zugriff auf übergebene Kommandozeilenparameter. Normalerweise wird ein Programm mit einem Befehl "Programm.exe" ausgeführt. Kommandozeilenparameter können daran angehängt werden und somit dem Programm zur Verfügung stehen: "Programm.exe -hallo -welt".

Der erste Parameter, `argc`, enthält die Anzahl der an das Programm übergebenen Parameter. Diese ist immer mindestens `1`, weil der Programmname immer der erste Parameter ist. Im Beispiel von oben wäre "Programm.exe" also 1, "-hallo" 2 und "-welt" 3.

Der zweite Parameter enthält logischerweise die Inhalte der Kommandozeilenparameter. `argv` ist ein Feld von Zeigern. Weil Zeiger und Felder erst später erklärt werden, solltest du zuerst bis dahin weitermachen und dann noch mal hier herblättern.

Da wir jetzt noch keinen richtig geeigneten Verwendungszweck haben, genügt es erst einmal, die Parameter auszugeben. Dev-C++ bietet eine Funktion, um Parameter beim Ausführen mit zu übergeben: Menu Debug -> Parameters. Hier kannst du (im oberen Feld) Parameter eingeben, jeweils mit Leerzeichen dazwischen.

Beispiel funktion6 - zudem wurden 2 Funktionen mit jeweils 2 Parametern implementiert.

Derartige Funktionalität lässt sich gut bei Programmen, die simple Dateiarbeiten machen, gut einsetzen. Tatsächlich lassen sich einige Packprogramme per Konsole bedienen.

Zusammenfassung

Nun ist auch dein Wissen zu Funktionen fast komplett. Ich hoffe sehr, dass du das Wesentliche nachvollziehen kannst - auch hier heißt es wieder üben, üben, üben!!! Mach

anschließend, wenn du alles gepeilt hast, ein kleines Rundenspiel mit einem Barbar (etwa Conan als Name), der sich verschiedene Waffen greifen und damit in den Kampf ziehen kann, etwa gegen ein Super-Ultra-Rhinozeros-Ork oder einen Tentakel-Ochsen-Frosch. Es sollte ausgewogen sein, du sollst auch deine eigene selbstkreierte künstliche Intelligenz beobachten können. Hier geht es zum ersten mal darum, ein größeres Projekt aufzubauen und zu leiten. Wenn du Bock hast und ein Kumpel auch, dann könnt ihr zusammen ein krasses Game entwickeln, das die Welt noch nicht gesehen hat!

Die Themen hier im zweiten Teil zu Funktionen haben damit zwar nicht sehr viel am Hut, aber egal. Um nun aber wieder zu den Funktionen zurückzukommen.

Workshop

Fragen

1.) Was ist kurz gesagt eine rekursive Funktion??
2.) Wird eine Funktion unbedingt `inline`, wenn man das so will??
3.) Was ist ein Feld aus Zeigern??

Programme

1.) Versuche, eine Funktion zum Potenzieren iterativ und rekursiv zu schreiben. Für Exponenten sollen nur ganze Zahlen vorgesehen sein.
2.) Schreibe zur Fibonacci-Funktion noch eine weitere Funktion, die einen Parameter `m` hat. Es soll die größte Fibonacci-Zahl herausgefunden werden, die kleiner oder gleich `m` ist.

Beispiel für einen Rundenkampf - **Globlok der Ork**.

Dieses erste komplexe Beispiel zeigt dir, wie du ein Spiel gestalten kannst. Versuch doch mal, das Programm zu verstehen - wenn es dir sehr schwer fällt, solltest du noch ein bisschen üben. In diesem Beispiel werden recht große Funktionen und teilweise schwer zu verstehende Bedingungen verwendet. Wenn du darin schon geübt bist, solltest du die Bedingungen interpretieren oder übersetzen können. Eigentlich genau die richtige Übung, um C++ zu verstehen.

Links:

http://www.volkard.de/vcppkold/rekursion.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop7.html
http://www.schornboeck.net/ckurs/ueberladung.htm
http://www.schornboeck.net/ckurs/default_param.htm
http://www.schornboeck.net/ckurs/rekursion.htm

Kapitel 8

Nun geht es etwas weiter bei den Variablen. Wir lernen andere Arten von Variablen kennen - nämlich Zeiger und Referenzen. Die sind uns dann bei Funktionen ganz nützlich, denn damit können wir Speicher direkt manipulieren. Vielleicht wird es dir nicht auf Anhieb klar, warum Zeiger an einer bestimmten Stelle verwendet werden. Das wirst du aber noch mitkriegen auf deiner Programmierer-Karriere.

Zeiger

Kurze Wiederholung aus Kapitel 2:

Speicher

Speicher gibt es in absolut jedem Computer. Er dient natürlich dazu, sich Daten/Informationen zu merken. Variablen (und ein Teil von Konstanten) stellen solche Daten/Informationen dar. Dazu legt der Computer an einem bestimmten Ort - der Adresse - diese Informationen ab. Ein Programm mit Variablen muss sich zwangsläufig darum kümmern, dass das System bzw. das Betriebssystem ihm den erforderlichen Speicher zur Verfügung stellt.

Ein Computer kann immer nur so viel Speicher adressieren - d.h. ansprechen - wie er konzipiert ist. Ein 16-Bit System/Betriebssystem kann nur 65536 Byte adressieren - gerade mal ~65 KByte. Mit 32 Bit-Systemen ist schon einiges mehr möglich, denn ein solcher Computer kann bis 4294967296 Byte ansprechen - das entspricht 4 Gigabyte Speicher. Oft sind Mainboards aber auf 2 oder 3 Gigabyte begrenzt wegen fehlender RAM-Slots. Kommende bzw. derzeitige 64-Bit-Monsterrechner könnten theoretisch bis 2^{64} Byte, also 18446744073709551616 Byte oder 16 Terabyte ansprechen!!!

Die Adresse kann so angegeben werden (auf ein 32 Bit-System bezogen): An Speicherstelle **01010101 10101010 00110011 01101100** ist der Wert **01101011** gespeichert. Das ist ein bischen umständlich und deswegen nimmt man einfach andere Zahlensysteme. Für Adressen hat sich das Hexadezimalsystem etabliert und für die Werte nimmt man das Dezimalsystem. In diesem Beispiel würde an der Adresse **55 AA 33 6C** der Wert **107** stehen.

Normale Variablen dienen dem Hauptzweck, Werte zu speichern. Mit Zeigern ist das etwas anders: Zeiger beinhalten Adressen aus dem Speicher eines Computers. Die Größe in Byte einer Variable hängt immer davon ab, welchen Datentyp du einsetzt, während Zeiger immer so groß sind, wie das System unterstützt.

Wie oben im Kasten gezeigt, hat ein 32 Bit-System bis 2^{32} Byte an Speicher zur Verfügung. Es ergibt sich, dass ein Zeiger immer ein 4-Byte Typ ist. Unabhängig davon, auf was für einen Datentyp er zeigt. Jetzt geht es erst einmal los mit Adressen...

Der Adressoperator &

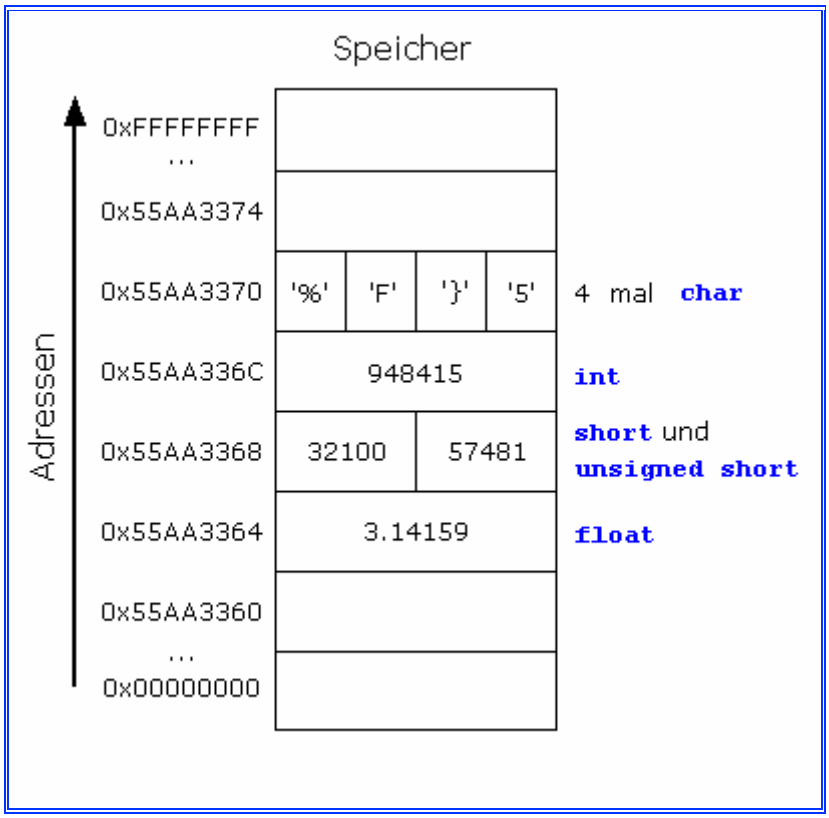
Zwar geht es bei jeder Variable hauptsächlich um den Wert, die Adresse ist aber trotzdem leicht herauszufinden. Eine ganz normale Variable mit einem & davor bedeutet, dass man die Adresse haben will:

```
int a = 600;
cout << "Adresse von a : " << &a;
```

Der Compiler lässt mit `cout` die Adresse von `a` hexadezimal ausgeben. Aber es gibt nicht so viele Stellen aus wie erwartet, weil die DOS-Box eben nur einen begrenzten Speicherbereich zur Verfügung hat.

Beispiel adress1.

Dass eine Variable eine Adresse bekommt, braucht dich nicht zu kümmern, das macht schon der Computer. Folgende Grafik zeigt, wie du dir einen Speicherausschnitt vorstellen kannst:



Dieser Streifen zeigt den Speicher wie in einer üblichen 32-Bit-Darstellung. Jede Variable hat eine Adresse, auch wenn es hier nicht den Anschein hat. Da es 4 Byte "breit" ist, können maximal 4 Variablen darin Platz finden. Optimal, um **`long`** und **`float`** darzustellen. Wie gesagt, dient eine solche Grafik nur zur Veranschauung.

Jedes mal, wenn der Wert einer Variablen angefordert wird, schaut der Computer an der Adresse der Variablen nach und holt (oder verändert) den Wert - die Adresse ändert sich dabei natürlich nicht.

Deklaration und Initialisierung eines Zeigers

Einen Zeiger zu deklarieren ist zunächst kaum etwas anderes als eine normale Variable zu deklarieren. Der einzige Unterschied im Code macht sich durch ein Sternchen `*`, vor den Namen gerückt, bemerkbar:

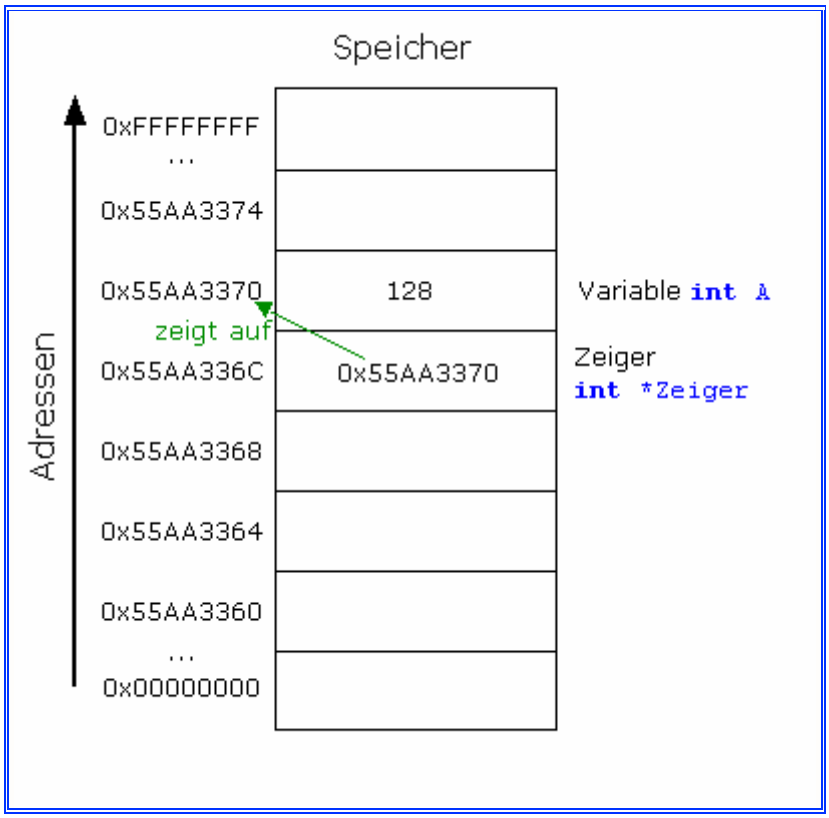
```
int *Zeiger = 0;
```

Da in einem Zeiger eine Adresse gespeichert wird, muss (oder sollte) dieser gleich einen Wert mit auf den Weg bekommen. Entweder tust du das mit NULL oder mit dem oben gelernten Adressoperator. Es ist kein guter Programmierstil, einen Zeiger nicht zu initialisieren und führt bei Zugriff auf die Adresse zu unkontrollierten Speicherzugriffen.

Im Speicher steht meist noch die eine oder andere Information, du kannst dir denken, dass der Zeiger einen unwillkürlichen Wert beim Deklarieren bekommt.

```
int A = 128;
Zeiger = &A; //bei einer Zuweisung sollte kein Sternchen vor dem Zeiger stehen;
// der Compiler weiß schon, dass es ein Zeiger ist
```

Zeiger zeigt also auf A. Den Speicher kannst du dir jetzt so vorstellen:



Zeiger hat also den "Wert" **0x55AA3370** und der ist an der Adresse **0x55AA336C** gespeichert. An der Adresse, auf die Zeiger zeigt, steht der Wert **128**. Um an die Adresse von Zeiger ranzukommen bzw. auszugeben, müsstest du schreiben:

```
cout << "Wo Zeiger gespeichert ist : " << &Zeiger;
```

Mit anderen Variablen:

Beispiel zeigerdeklaration1.

Benennung von Zeigern

Für Zeiger gelten bei der Benennung genau die selben Regeln wie bei den normalen Variablen. Oft wird ein kleines p vorne drangefügt, um zu verdeutlichen, dass es sich um einen Zeiger handelt.

Zeiger dereferenzieren

Um auf den Wert zuzugreifen, der an der Adresse gespeichert ist, auf die der Zeiger zeigt, gibt es den Dereferenzierungsoperator *. Den gibt es allerdings auch schon als Mal-Zeichen und als Zeichen für eine Zeigerdeklaration, also nicht verwechseln!!

```
*Zeiger = 256;
cout << A;
```

A ist jetzt 256. Der Wert an der Adresse, auf die Zeiger zeigt, ist von Zeiger mittels Dereferenzierungsoperator geändert worden.

Freilich könnte man das auch direkt machen:

```
int A = 128;
A = 256;
```

Aber mit einem Zeiger auf eine Variable kann man diese Variable in einer Funktion verändern, ohne dabei einen Wert zurückgeben zu müssen.

Beispiel zeiger1.

Zeiger in Funktionen

Beispiel zeiger2.

Hier wurde es umgangen, Argumente zu übergeben, indem einfach ein globaler Zeiger deklariert wurde. Es geht allerdings auch so:

Beispiel zeiger3.

Da hier mehrmals die gleichen Operatorzeichen gebraucht werden, zeige ich noch einmal die Unterschiede:

Operator	Bedeutung	Beispiel mit Position
&	Bitweises UND - verknüpft 2 Werte miteinander	`int a = 12 & 15;int b &= 5;`
&	Adressoperator - macht die Adresse zugänglich	`int *c = &a; cout << &b;`
*	Arithmetischer Operator für Multiplikation	`int d = 5 * 12;`
*	Stern - gibt an, dass es sich um einen Zeiger handelt	`int *e = 0;`
*	Dereferenzierungsoperator - macht den Wert an einer Adresse zugänglich	`*e = 1;`

Das Schöne daran ist natürlich, dass weniger globale Variablen nötig sein werden. Parameter sind immer Kopien ihrer Argumente. Wenn Werte übergeben werden, gibt es zwei Speicherstellen, die voneinander unabhängig sind. Eine übergebene Adresse jedoch bedeutet immer Zugriff auf die gleiche Speicherstelle.

Das gröbste von den Zeigern ist nun eigentlich verstanden. Nimm dir Zeit, die Operatoren in Fleisch und Blut übergehen zu lassen. Zeiger werden in kommenden Kapiteln verdammt oft gebraucht!!!!!!!!!!

Zeiger auf Konstanten

Nein, Zeiger können nicht nur auf Variablen zeigen, sondern auch auf Konstanten. Der Computer lässt sich aber durch keinen Zeiger austricksen - Konstanten werden selbst durch Zeiger nicht variabel.

```
const Datentyp *Name = &Konstante;
```

In Anwendung:

```
const float PI = 3.14159f;
const float *Konstantenzeiger = &PI;

//PI = 0.815f;
//*Konstantenzeiger = 2+3; //geht beides nicht!!
```

Konstante Zeiger

Solche zeigen immer auf die selbe Adresse, der Wert an der Adresse kann aber geändert werden. Im Vergleich zu den Zeigern auf Konstanten hat sich nur die Position des **const** geändert:

```
Datentyp *const Name = &Variable;
```

Kleines Beispiel:

```
int Mhz = 2400;
int Mhz2 = 3000;
float *const Penzjum4 = &Mhz;

//Penzjum4 = &Mhz2; //nur das geht nicht!!
*Penzjum4 = 2400+200;
```

Bedingung ist aber wie im Beispiel, dass die Variablen, auf die der Zeiger zeigen soll, nicht konstant sind.

Beispiel zeiger4.

Zeiger auf void

Jetzt siehst du **void** in einem anderen Einsatz als bei Funktionen. Mit einem Zeiger des Typs **void** kannst du auf alle anderen Variablen, egal von welchem Typ sie sind, zeigen lassen. Du kannst sie aber nicht direkt einer "reinen" Variablen zuweisen, sondern musst mit dem Type-Cast und dem Dereferenzierungsoperator den Wert zugänglich machen:

```
int A = 33;
void *Universal = &A;
*(static_cast<int*>(Universal)) = 188;
```

Diesen Konstrukt musst du anwenden, da der Compiler sonst nicht weiß, was mit dem Zeiger gemeint ist. Wir veranlassen den Compiler sozusagen, den **void**-Zeiger als anderen Zeiger - hier **int*** - anzusehen. A hat den neuen Wert 188 bekommen.

Beispiel zeiger5.

Zeiger auf Funktionen - "Funktionszeiger"

Damit kann man komfortabel zwischen verschiedenen Funktionen "hin- und herschalten". Die Funktionen, auf die der Zeiger zeigen kann, müssen dabei die selben Parameter und Rückgabetypen haben. Um einen Funktionszeiger zu deklarieren, musst du folgendes schreiben:

```
Rückgabetyp (*Name)(Parametertyp);
```

Um diesen Zeiger auf eine Funktion zeigen zu lassen, schreibst du folgendes:

```
void (*FZ)(int); //als Beispiel für eine Zeigerdeklaration
void Funktion1(int Ganzzahl);

FZ = Funktion1; //ohne Klammern hintendran!!
int Hallo = 26;
FZ(Hallo);
```

Ohne den FZ-Funktionszeiger hättest du schreiben müssen (wie du es sonst getan hättest):

```
Funktion1(Hallo);
```

Zeiger auf Zeiger

Des weiteren gibt es in C++ die Möglichkeit, Zeiger auf Zeiger zu erstellen (dies wirst du aber eher irgendwo finden als selbst anwenden). Die Deklaration eines Zeigers auf einen Zeiger siehst du hier:

```
int **ZZ;
```

Nun weiß der Compiler, dass es einen Zeiger auf einen anderen Zeiger gibt, namens ZZ. Dieser soll jetzt auf einen Zeiger Z zeigen, wobei Z auf die Variable Zahl zeigt:

```
int Zahl = 2332;
int *Z = &Zahl; //Z zeigt auf Zahl
ZZ = &Z;        //ZZ zeigt auf Z
```

Zahl ist eine Variable, die den Wert 2332 hat. Zeiger Z zeigt auf Zahl (anders gesagt: Zeiger Z hat den Wert, der die Adresse von Zahl ist). ZZ bekommt dann den Wert, der die Adresse von Z ist (wieder mittels des Adressoperators &). Wie sieht nun der Zugriff auf die Werte aus??

```
**ZZ *= 2;        //Zahl ist nun doppelt so groß

int Nummer = 3;
int *P = &Nummer; //P zeigt auf Nummer
*ZZ = &Nummer;    //Z zeigt auch auf Nummer
ZZ = &P;          //ZZ zeigt nun auf P
```

Wie gesagt ist diese Technik selten in Gebrauch. Wenn es um dynamischen Speicher geht, kommen solche Zeiger zum Einsatz, um mehrdimensionale Felder dynamisch anzulegen. Solltest du sie gebrauchen, so kommentiere bitte ausreichend!

Referenzen

Eine Referenz ist eine Variable, die eine andere Variable "nachmacht"; sie hat immer exakt den selben Wert wie die Variable, auf die sie zeigt. Sie wird zum Synonym der Variable - ändert sich die eine, so ändert sich auch die andere. Du kannst dir das wie ein sich selbst dereferenzierender Zeiger vorstellen.

Beim Deklarieren unterscheidet sich eine Referenz von einem Zeiger nur in einem kaufmännischen Und. So sieht's aus:

```
VariablenTyp & VariablenName = AndereVariable;
```

Diese Referenz muss direkt bei der Deklaration initialisiert werden, sonst weiß der Compiler nicht, auf welche Variable die Referenz zeigt.

Beispiel reference1.

Ist eine Referenz einmal einer Variable verschrieben, so gibt es für die Referenz kein Zurück mehr. Du kannst nach der Initialisierung die Referenz keine andere Variable heiraten lassen :)

Referenzen auf Referenzen kannst du nicht so erstellen wie Zeiger auf Zeiger. Folgendes Beispiel:

```
int Zahl = 5;
int Zahl2 = 7;

int *Z = &Zahl;
int &R = Zahl2;

int **ZZ = &Z;
//int &&RR = R; //geht nicht
int &RR = R;    //geht
```

`Zahl2`, `R` und `RR` sind nun ein und dieselbe Variable.

Übergabe per Referenz

Du kannst mithilfe von Referenzen die Kopien der übergebenen Variablen einer Funktion gleich ins Hauptprogramm übertragen:

```
void Function(int a, int &b)
{
   a *= 33;
   b *= 12;
   cout << a << '\n' << b;
}

int main(void)
{
   int X = 3;
   int Y = 4;

   cout << X << '\n' << Y; // 3 4
   Function(X, Y) ;        //99 48
   cout << X << '\n' << Y; // 3 48
```

```
    cin >> X;

    return 0;
}
```

Im Prototyp der Funktion ist `b` eine Referenz, das heißt, dass alle Änderungen innerhalb der Funktion sich sofort auf die Hauptfunktion auswirken. Deshalb ist `Y` nach dem Funktionsaufruf weiterhin verändert, während von `X` nur die Kopie verändert wird - und die geht am Ende von `Function()` verloren.

Übergabe mit Referenzen wird auch als "Call by Reference" bezeichnet. Und wie ich es weiter oben schon erklärt habe, bezeichnet "Call by Reference" auch die Übergabe von Adressen mit Zeigern und Adressoperatoren. Wenn ein Parameter eine Referenz ist, wird sie auf ihr Argument referenzieren, wenn sie gebraucht wird.

Bei "Call by Reference" musst du darauf achten, dass du wirklich Variablen verwendest. Keine literalen Konstanten und keine mit `#define` definierte Konstanten!

Zusammenfassung

Dieser Kapitel hatte es mal wieder in sich. Zeiger sind für viele Neulinge zunächst ziemlich schwer zu verstehen. Aber es lohnt sich durchaus, sich damit zu befassen.

Zeiger sind Variablen, die dafür vorgesehen sind, Adressen zu speichern. Jede Variable hat eine Adresse im Speicher - die ist mittels des Adressoperators `&` herauszufinden. Eine Adresse kann dann dem Zeiger zugewiesen werden.

Der Dereferenzierungoperator `*` dient einem Zeiger dazu, auf den Wert, der an der gespeicherten Adresse steht, zuzugreifen. Einen Zeiger deklariert man auch mit dem Operator, ebenso wie man mit `*` Werte multipliziert. Anfangs könnte es dir Schwierigkeiten bereiten, diese Operatoren auseinanderzuhalten.

Referenzen sind Kopien von anderen Variablen. Darüber hinaus ändert sich ihr Wert, wenn der Wert der Variable sich ändert und andersrum. Eine Referenz kannst du mit einem `&` vor dem Namen deklarieren; du musst sie dann auch gleich initialisieren. Dazu muss die gewünschte Variable bereits vor der Referenz existieren.

Innerhalb einer Funktion kannst du auf Variablen, die eigentlich vom Parameter her kommen, zugreifen. Einfach Zeiger oder Referenzen als Parameter nehmen. Nachteil ist, dass du beim Funktionsaufruf extra eine Variable für den Parameter da haben musst.

Workshop

Fragen

1.) Was ist einfach gesagt ein Zeiger??
2.) Was beinhaltet ein Zeiger und eine Referenz??
3.) Was kann man bei Zeigern und Referenzen ändern??
4.) Was gibt es bei Zeigern mit **`const`** zu beachten??
5.) Was gibt `&Zeiger`, was `Zeiger` und was `*Zeiger` an??
6.) Wieso gibt es unterschiedliche Werte bei `&Zeiger` und `Zeiger`??
7.) Wieso gibt es Zeiger auf Zeiger??
8.) Wieso gibt es kein "Call by Pointer"??

Links:

http://www.volkard.de/vcppkold/zeiger.html http://www.volkard.de/vcppkold/referenzen.html http://www.volkard.de/vcppkold/call_by_value.html http://www.volkard.de/vcppkold/call_by_reference.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop6_1.html http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop6_4.html
http://www.schornboeck.net/ckurs/referenzen.htm http://www.schornboeck.net/ckurs/zeiger.htm

Kapitel 9

Komplexe Datentypen

Komplexe Datentypen (schon mehrmals zuvor erwähnt) sind zusammengeschusterte/-gesetzte Datentypen. Einfache Datentypen beinhalten immer nur einen Wert, während komplexe eine bestimmte Anzahl an Werten speichern kann. Hier lernst du Felder und Strings kennen und im Kapitel darauf werde ich dir Strukturen, Aufzählungstypen, Unionen und Namensbereiche erklären.

Statische Felder (oder Arrays)

In dem Beispiel **GDO** aus Kapitel 4 gibt es eine Menge von Variablen - für jeden Arm gibt es Schaden1, Schaden2 ... und Waffe1, Waffe2 usw. Jede dieser Variablen hat einen eigenen Namen. Mit Feldern wollen wir diesen etwas unschönen Programmcode besser in den Griff bekommen.

Ein Feld ist in C++ eine Sammlung von Variablen mit dem gleichen Datentyp. Da dieses eine bestimmte Anzahl an Variablen haben soll, muss das dem Compiler auch klar gemacht werden.

Felder deklarieren

Du musst, um ein Feld zu deklarieren, den Datentyp, den Namen und dann die Anzahl der Elemente in eckige Klammern schreiben.

Verallgemeinert sieht das dann folgendermaßen aus:

```
VariablenTyp FeldName [ AnzahlDerElemente ];
```

Der Variablentyp kann einer der bereits bekannten (**short**, **int**, **char**, **bool** ...) oder ein komplexer Datentyp sein. Da wir aber noch keine anderen komplexen Datentypen kennen, sind wir dazu gezwungen, die einfacheren Wesen zu verwenden.

Für den Namen gelten wie immer die Regeln für Variablennamen. Auf den Namen folgen dann eckige Klammern - darin ist die Anzahl der Feldelemente (so heißen die einzelnen Variablen im Feld) festgelegt; das kann entweder über eine Zahlenkonstante (`1`, `2`, `4`, `39`...), über einen Term (`1*5`, `99+66`, `A+2*B`), oder über eine Variable erfolgen. Dabei ändert sich die Anzahl nicht, wenn die Variable geändert wird; bei einem statischem Feld kann man die Anzahl der Feldelemente nicht während des Programms ändern.

Auf Feldelemente zugreifen

Um auf ein Element im Feld zugreifen zu können, musst du angeben, auf welches der vielen Elemente du zugreifen willst. Das passiert ebenfalls mit eckigen Klammern. Allerdings musst du beachten, dass das erste Element die Zahl `Null` in den Klammern hat. Demzufolge kannst du auf das letzte Element zugreifen, indem du einfach den Wert, mit dem das Feld deklariert wurde, um 1 erniedrigst.

```
#define AnzahlDerElemente 100
#define letztes AnzahlDerElemente - 1

int FeldName[AnzahlDerElemente];
FeldName[12] = 33;
FeldName[0] = 1;
FeldName[letztes] = 9;
```

Beispiel feld1 zeigt die Deklaration und Zugriff auf ein Feld.

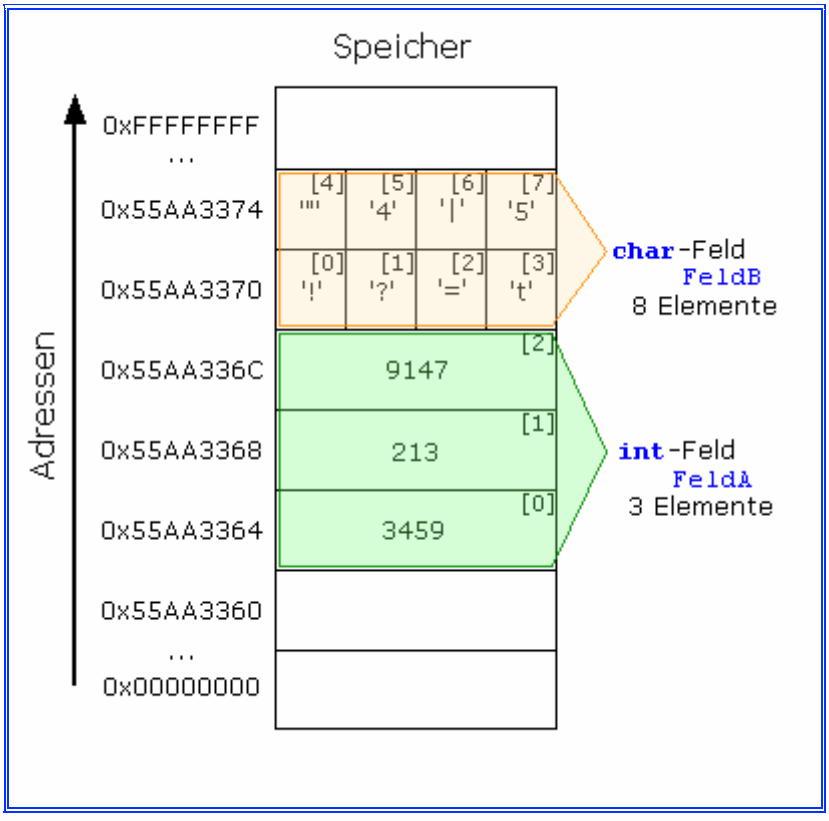
Die Sache mit der Null als Index für das erste Element ist zunächst noch ein bisschen verwirrend. Es gibt sogar einen Begriff dafür, auf das Element nach dem letzten Index zuzugreifen. Man nennt das "Fence Post Error". Der Begriff - zu deutsch "Zaunpfahlfehler" - entstand aus der Frage, wie viele Zaunpfähle man brauchen würde, um einen 10m langen Zaun aufzustellen, bei dem nach jedem Meter ein Pfahl steht. Vorschnell geantwortet sind es 10, mit richtiger Überlegung werden aber 11 draus.

Wie Felder im Speicher aussehen

Wenn du ein Feld deklarierst, reserviert der Computer für x Elemente Speicher. Es entsteht eine Kette von Variablen des genannten Datentyps. Für folgende Deklarationen

```
int FeldA[3];
char FeldB[8];
```

können wir folgenden Speicherinhalt erwarten:



Die darinstehenden Werte sind zufällig noch an den Speicherstellen gewesen. Um ein Feld von Anfang an sinnvoll nutzen zu können, solltest du es immer initialisieren, wie im folgenden Abschnitt gezeigt.

Felder initialisieren

Variablen werden normalerweise mit Werten belegt, die zuvor im Speicher standen. Deswegen ist es sinnvoll, sämtliche Elemente zu initialisieren. Du kannst das tun, indem du

- jedes einzelne Element mit einem Wert belegst, oder
- jedes einzelne Element mittels einer Schleife zu einem Wert verhilfst, oder
- die Werte gleich bei der Deklaration in geschweifte Klammern hintenan setzt.

Die ersten beiden Möglichkeiten kannst du dir selber zusammenreimen. Dei dritte Möglichkeit sieht so aus:

```
int FeldName[10] = { 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 };
```

Der Compiler weist dann jedem Element den Wert an der zugehörigen Stelle zu. Element **0** hätte den Wert 1, Element **1** den Wert 3, Element **2** den Wert 6 und so weiter. Hier musst du für alle Elemente einen Wert in die Klammern setzen.

Du kannst bei einer solchen Initialisierung auch die Angabe in der eckigen Klammer weglassen:

```
int FeldName[] = { 128, 256, 512 };
```

Richtig erkannt, dieses Feld hat drei Elemente. Der Compiler zählt die Werte in den geschweiften Klammern und macht das Feld entsprechend groß genug.

Wie groß ein Feld ist

Je nachdem, was für ein Datentyp verwendet wird und wie groß das Feld ist, wird Speicher im Arbeitsspeicher reserviert. Die Größe in Byte ist dabei das Produkt aus Anzahl der Elemente und Größe des Datentyps. Um die Größe eines Datentyps oder eines Feldes herauszukriegen, gibt es in C++ das Schlüsselwort **sizeof()**:

```
int FeldName[] = { 1, 2, 3 };
int GroesseVomFeld = sizeof(Feldname) / sizeof(int);
```

Die Größe einer **int**-Variable ist (in einer 32 Bit-Umgebung wie Windows) 4 Byte; Die Größe des Feldes ist 3 * 4 Byte, weil es aus drei **int**-Variablen besteht.

Indexberechnung von Feldern

Für jedes Feld wird immer nur die Adresse des ersten Elementes gespeichert. Die restlichen Adressen werden immer live berechnet. Der Computer berechnet diese, indem er die Adresse des ersten Elementes zu der Größe des verwendeten Datentyps mal den Index rechnet:

```
int Feld[5];
//Feld[0] ist bei Adresse 0x1000FF30
//Feld[1] ist bei Adresse 0x1000FF34 weil eine int-Variable 4 Byte
groß ist
```

Die Adresse des ersten Feldelementes wird in einem Zeiger gespeichert. Er hat exakt den Namen des Feldes, nur eben ohne die Indexangabe. Für das obige Beispiel würde der Zeiger einfach "Feld" heißen. Ein Ausdruck

```
int a = *Feld;
```

würde a zu genau dem gleichen Ergebnis verhelfen wie

```
int a = Feld[0];
```

Weil das so ist, kann ein Feld auch ohne Probleme als Argument für einen Zeiger-Parameter verwendet werden.

Mit dem Zeiger ist auch noch einiges weiteres möglich:

Zeigerarithmetik

Nun wissen wir, dass ein Feld auch ein Zeiger sein kann. Mit Zeigerarithmetik können wir diesen Zeiger sogar für das ganze Feld benutzen. Anstatt den Index in die Klammern zu schreiben, können wir den Zeiger in einem Ausdruck um den Index erhöhen.

```
int Feld[5];

Feld[0] = 1;
*Feld *= 2;
cout << "Feld Element 1 : " << Feld[0];

Feld[1] = 2;
*(Feld + 1) *= 2;
cout << "\nFeld Element 2 : " << Feld + 1;

Feld[2] = 3;
*(Feld + 2) *= 2;
cout << "\nFeld Element 3 : " << Feld + 2;

Feld[3] = 4;
*(Feld + 3) *= 2;
cout << "\nFeld Element 4 : " << Feld[3];

Feld[4] = 5;
*(Feld + 4) *= 2;
cout << "\nFeld Element 5 : " << Feld + 4;
```

Dieser Code kompiliert würde allen Feldelementen einen Wert zuweisen, diesen verdoppeln und anschließend ausgeben. Wie du siehst, ist es nicht nötig, die exakten Bytegrößen zu verwenden. Der Compiler nimmt einfach den hinzuzufügenden Wert und verwendet ihn als Index - und der muss ja auch nicht den exakten Bytegrößen entsprechen.

Klar ist, dass dieser Index die Feldgrenzen nicht über- und nicht unterschreiten darf. Stell dir vor, ein gigantisches Feld würde deinen ganzen Arbeitsspeicher beanspruchen; wie würde der Rechner wohl reagieren, wenn du ihn auf ein Element außerhalb deines RAMs zugreifen lassen würdest??

Beispiel feld2.

Felder kopieren

Weil die Felder ihre Adresse als Zeiger gespeichert haben, ist es nicht möglich, ein Felder einem Anderem direkt zuzuweisen (`Feld1 = Feld2;`). Das würde bedeuten, dass zwei Felder die gleiche Speicheradresse haben, und das geht nicht. Du kannst das aber lösen, indem du jedem einzelnem Wert den anderen Wert zuweist - wie im Abschnitt "Felder initialisieren". Am elegantesten geht es natürlich mit einer Schleife:

```
int Feld1[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int Feld2[10];

for(int i = 0; i < 10; i++)
{
   Feld2[i] = Feld1[i];
}
```

Wenn die beiden Felder unterschiedlich groß sind, musst du das natürlich so regeln, dass bei keinem Feld auf Speicher außerhalb zugegriffen wird.

Felder aus Zeigern

Anstatt normale Variablen zu verwenden, kannst du auch Felder aus Zeigern oder Referenzen erstellen.

```
int Feld[5] = { 1, 3, 5, 7, 9 };
int *Feld2[5];

for(int i = 0; i < 5; i++)
{
   Feld2[i] = &(Feld[i]);
}
```

Diese Zeiger enthalten wie gewöhnliche Zeiger Adressen. Oben siehst du, wie man an die Adresse eines Feldelementes außer über Zeigerarithmetik rankommt. Den Wert an der gespeicherten Adresse erhälst du mit dem Dereferenzierungsoperator - du solltest aber Klammern so setzen, dass es leicht als Dereferenzierung erkennbar ist:

```
*(Feld2[2]) = 2;
```

Beispielprojekt dazu ist im folgenden Abschnitt.

Die 4.te Dimension

Bis jetzt waren alle Felder eindimensional. Es geht aber auch, dass Felder mehrere Dimensionen beinhalten. Das lässt sich einrichten, indem du beliebig viele eckige Klammern nach dem Namen setzt:

```
int Feld[8][8];//zweidimensionales Feld
int
Feld2[1888][221321][13123123][123123123][13232][12323][3232312][12312312][1
3123123][12313123];
//Es gehen unvorstellbar viele Dimensionen (begrenzt durch des Computers
Speicher)!!
```

Um derartige Felder zu initialisieren, musst du jede einzelne Dimension initialisieren. `Feld2` hätte damit wohl große Probleme.

`Feld` würde 8·8 also 64 Byte groß sein. Wie es bei `Feld2` steht, weiß ich nicht - Ich bezweifle stark, dass dieses Feld in einen 4 Gigabyte Arbeitsspeicher passen könnte. Es ist eher ein negativ-Beispiel, wie man es nicht machen sollte.

Beispiel feld3.

In der Praxis kommen meistens höchstens zweidimensionale Felder vor. Felder mit mehr als 3 Dimensionen kann man sich nicht so richtig vorstellen; wie stellst du dir ein dreidimensionales Feld vor?? Wie einen Würfel natürlich. Und wie ein vierdimensionales??

Strings

Bis jetzt waren alle Zeichenketten konstant, weil sie direkt im Quellcode standen, also während des Programms nicht änderbar. Mit Feldern ist uns zunächst eine Möglichkeit gegeben, das auszubessern. Variable Zeichenketten kannst du erzeugen, indem du ein Feld aus **char**-Variablen anlegst:

```
char String[] = {88,89,90,0}; // ... = {'X', 'Y', 'Z', '\0'};
String[0] = 65;
String[1] = 66;
String[2] = 67;
```

Die erste Zeile ist schon ein ziemlich umständlich (und durchaus fehleranfällig), das macht sich vor allem bei langen Zeichenketten bemerkbar. Schneller geht es mit den schon verwendeten Zeichenkettenkonstanten:

```
char String[] = "XYZ";
```

Dir ist wahrscheinlich aufgefallen, dass der String mit der einzelnen Zuweisung noch eine Null hinten dran gefügt wurde, die konstante Zeichenkette dies aber nicht hat. Viele Funktionen benötigen das Null-Zeichen, um das Ende des Strings zu identifizieren.

Das obere Exempel bleibt vom Compiler ungeändert - die konstante Zeichenkette wird aber beim Compilern angeguckt und das Null-Zeichen wird dazugefügt, wenn der String die Null noch nicht hat. Bei der Ausgabe

```
cout << sizeof(String);
```

würde für beide Strings 4 rauskommen. Um die Größe des wahren Strings zu erfahren, müsstest du folglich `sizeof(String) - 1` rechnen.

Beispiel string1.

In diesem Beispiel wird noch eine andere Sache gezeigt, nämlich die Eingabe eines solchen Strings. Wenn du nur ein Wort eingibst, ist alles normal, wenn du aber ein Leerzeichen und anschließend ein weiteres Wort eingibst, beendet das Programm. Das allerdings liegt in der Eingabefunktion `cin` begründet. Deshalb solltest du hier immer nur ein Wort eingeben. Wir werden später noch eine bessere Möglichkeit kennenlernen, um dann auch mal ganze Sätze eingeben zu können.

Funktionen für Strings

Beispiel string2.

In der Header-Datei **cstring** gibt es eine Reihe von Funktionen, die den Umgang mit Strings ein wenig vereinfachen.

strlen()

`strlen()` (von string lenght) gibt an, wie viele Zeichen einen Wert haben, bis (irgend)ein Null-Zeichen kommt. Verwechsel das nicht mit der Größe des Feldes.

Das Null-Zeichen selbst beachtet `strlen()` dabei nicht. Die Funktion hat den folgenden Prototypen:

```
size_t strlen(const char *String);
```

`size_t` ist ein vorzeichenloser Ganzzahltyp, der Größeninformationen enthalten soll.

strcpy()

Mit `strcpy()` (von string copy) wird der Inhalt eines Strings in einen Anderen kopiert. `strcpy()` hat folgenden Prototypen:

```
char* strcpy(char *Ziel, char *Quelle);
```

Voraussetzung ist, dass das **`Ziel`** mindestens genauso groß ist, wie die Quelle, sonst werden einige Buchstaben weggelassen. Das `Null`-Zeichen setzt `strcpy()` automatisch.

strncpy()

Diese Funktion erweitert `strcpy()` um die Möglichkeit, die Anzahl der zu kopierenden Zeichen besser zu kontrollieren. Ein zusätzlicher Parameter bestimmt die maximale Anzahl an Zeichen, die kopiert werden. Sie hat folgenden Prototypen:

```
char* strncpy(char *Ziel, char *Quelle, size_t MaxCopy);
```

Wieder so ein `size_t` (vorzeichenloser Ganzzahltyp).

strcat()

```
char *strcat(char *String, char *Anhang);
```

`strcat()` hängt den String `Anhang` an `String` an. Das `\0`-Zeichen aus `String` wird mit dem ersten Zeichen aus `Anhang` überschrieben. Zwischen `NULL`-Zeichen und dem Ende von `String` muss soviel Speicher frei sein, dass `Anhang` hineinpasst.

strcmp()

```
int strcmp(char *String1, char *String2); //vergleicht die beiden
Strings;
//wenn beide gleich sind, wird Null zurückgegeben
```

strcmp() vergleicht alle Zeichen der beiden Arrays `String1` und `String2`. Ein Großbuchstabe kommt alphabetisch nach seinem kleinen Äquivalent. Eine Zahl wird zurückgegeben, sobald das erste unterschiedliche Zeichen gefunden wurde oder die Strings vollkommen gleich sind:

- `< NULL` – `String1` ist kleiner als `String2`
- `= NULL` – `String1` ist genauso groß wie `String2`
- `> NULL` – `String1` ist größer als `String2`

Wenn du weitere Funktionen, die Strings arbeiten, kennen lernen willst, schau doch mal bei der Referenz der Funktionen vorbei. Folge dem Link **cstring**. Dort findest du viele Funktionen der C++-Funktionenbibliothek kurz erklärt.

Zusammenfassung

Du hast in diesem Kapitel gelernt, was Felder und Strings sind. Felder sind Ansammlungen von Variablen des gleichen Datentypes. Ein String ist eine Untermenge/ein Teil von Feldern - ein Feld aus **char**s. Wieder hatte es viel mit Zeigern und Adressen zu tun gehabt. Man könnte das Themengebiet auch ohne Kenntnis von Zeigern verstehen, aber dann würde man wahrscheinlich so manches mal Schiffbruch erleiden.

Wann immer du eine Reihe von Werten hast, die zu einem ähnlichen Zweck gebraucht werden sollen, ist es ratsam, Felder einzusetzen. Du hast gesehen, mit Schleifen kann man Felder effektiv verbinden. Für manche Aufgaben ist es sogar unerläßlich, Felder zu benutzen - Beispiel Schachfeld.

Wie schon gesagt, könntest du dich mit dem Speicher verhaspeln. Zu viele Dimensionen zu erstellen fördert auch die Unleserlichkeit.

Zum Schluss hast du noch einige Funktionen kennengelernt, um Strings zu bearbeiten. Auf jeden Fall solltest du viele Projekte programmieren, um mit Feldern und Strings gut klar zu kommen. Die Themen sind für viele Anfänger etwas verwirrend und können durchaus die heile Programmierwelt zum Wanken bringen.

Workshop

Fragen

1.) Wozu dienen die eckigen Klammern??
2.) Wieso beginnt der Index bei Null anstatt bei 1??
3.) Kann man auch einen negativen Index verwenden??
4.) Deklariere und initialisiere ein dreidimensionales Feld mit drei Elementen pro Dimension!
5.) Wieso ist ein Feld ein komplexer Datentyp, wenn es doch auch nur aus einfachen Datentypen besteht??

Programme

1.) Der Benutzer soll eine Zahl eingeben, die dann als Größe für ein Feld verwendet werden soll. Der Benutzer soll dann alle Elemente dieses Feldes mit Werten belegen. Zum Schluss soll ihm das Programm die Summe der Werte

ausgeben.
2.) Lass den User höchstens 20 Zahlen eingeben (bis 20 Zahlen erreicht sind oder 0 eingegeben wurde). Der Rest soll wie in Aufgabe 1 aussehen.
3.) Versuche die beiden Programme mal ohne Felder!

Links:

http://www.volkard.de/vcppkold/arrays.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop6_2.html
http://www.schornboeck.net/ckurs/array.htm

Kapitel 10

Hier geht es gleich weiter mit den komplexen Datentypen. Du lernst in diesem Kapitel Strukturen, Unionen, Aufzählungstypen und Namensräume kennen.

Strukturen

Während Felder Ansammlungen mehrerer Variablen des gleichen Typs sind, handelt es sich bei Strukturen in C++ um einen noch komplexeren Typ - Strukturen können mehrere verschiedene Datentypen beinhalten.

Eine Struktur deklarieren

Um eine Struktur zu erstellen, muss sie mit dem Schlüsselwort **`struct`** (von structure) deklariert werden, gefolgt vom Namen, der nach der Deklaration als eingens definierter Variablentyp genutzt werden kann:

```
struct Computer
{
   bool An;
   short Mhz;
   short RAM;
};
```

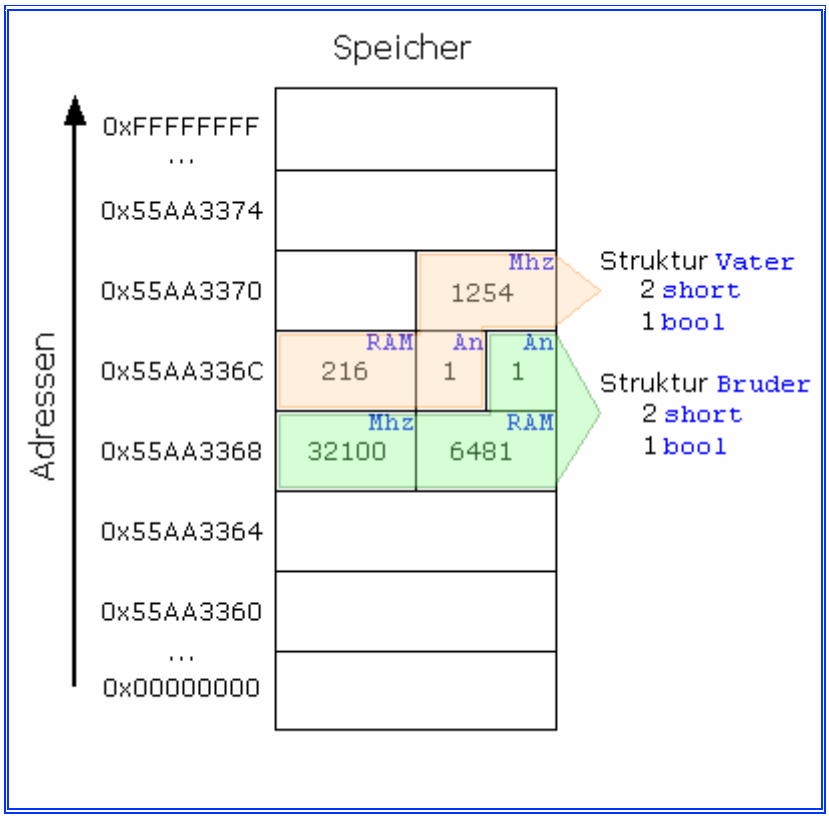
Dieser Code deklariert die Struktur. Das Semikolon hinter der schließenden Klammer ist absolut nötig. Es werden bei der Deklaration keine Variablen dieses Datentyps erschaffen, was aber eigentlich möglich wäre.

Instanzen erstellen

Der Begriff "Instanz" kommt eher aus der objektorientierten Programmierung. Bedeutet ziemlich das selbe wie "Variable". Eine Instanz zu instanzieren bedeutet, den Computer dazu anzuweisen, Speicher für die Instanz zu beschaffen. Bei Strukturen geht das nicht anders als bei den einfachen Datentypen:

```
Computer Bruder, Vater;
```

Für `Bruder` und `Vater` könnten dadurch beispielsweise folgende Speicherabschnitte reserviert werden:



Da den Elementen in den beiden Strukturen noch keine Werte zugewiesen wurden, nehmen sie vorerst die Werte an, die im Speicher stehen. Das ist bei Strukturen nicht anders als bei normalen Variablen.

In diesem Fall hat die Struktur die Größe 5 Byte. Es könnte auch sein, dass der Computer beim Speicher reservieren die einzelnen Elemente nicht direkt aneinander packt, also dass zum Beispiel `An` nicht gleich nach `RAM` folgt, wie das bei der Struktur `Bruder` ist. Die `sizeof(Struktur)`-Angabe könnte dann vom theoretischen Wert abweichen.

Weiter oben habe ich erwähnt, dass Instanzen auch sofort bei der Deklaration erzeugt werden können. Ich hätte also auch schreiben können:

```
struct Computer
{
   bool An;
   short Mhz;
   short RAM;
}Bruder, Vater;
```

Diese Definition von Strukturvariablen ist genau das gleiche wie die Möglichkeit, die aus 2 Anweisungen (so kann mans wohl nennen) besteht.

Wie du auf Elemente zugreifst

Bei einer Struktur kannst du (wie du dir wohl schon gedacht hast) auf jedes deklarierte Element zugreifen. Und damit du das tun kannst, musst du den Element-Auswahl-Operator '`.`' (einen ganz gewöhnlichen Punkt) verwenden:

```
Bruder.An = 0;
Bruder.Mhz = 1234;
Bruder.RAM = 2048;

Vater.An = 1;
Vater.Mhz = 536;
Vater.RAM = 31;

Vater.RAM = Vater.Mhz * Bruder.RAM + Bruder.Mhz;
```

Es sind alle Operationen mit den Elementen wie mit einer normalen Variable, wenn diese vom gleichen Typ wäre. Einzige Schwachstelle ist, dass bei längeren Namen die Übersichtlichkeit den Bach runtergeht.

Beispiel struct1.

Felder von Strukturen

Wenn Felder von einfachen Datentypen möglich sind, dann gibt es in C++ auch Felder von Strukturen. Wie schon bekannt sein dürfte, musst du die Feldstärke in eckigen Klammern angeben:

```
Computer ComputerInDerFirma[16];

//Zugriff:
ComputerInDerFirma[12].Mhz = 33;
ComputerInDerFirma[0].Mhz = 1533;
```

Du solltest immer im Hinterkopf behalten, dass eine Struktur eine Menge Speicher braucht. Hier sind die Strukturen zwar nicht so überwältigend groß, bei großen Projekten solltest du aber darauf achten, dass du nicht zu viel Speicher verschwendest. Irgendwann ist auch mal der größte Speicher erschöpft...

Strukturen in anderen Strukturen

Noch mehr Speicher verschwenden kannst du, indem du eine Struktur in eine andere reinpackst:

```
struct Firma
{
   short Mitarbeiter;
   short ComputerAnzahl;
   Computer ComputerDerMitarbeiter[16];
   short Umsatz;
}Diese, Napoleon, Computerverlag;

Diese.ComputerDerMitarbeiter[3].Mhz = 128;
```

Wenn du mehrere Firmen erstellst, hast du auch gleich eine Armee von Computern. Es ist zu bezweifeln, dass eine Firma namens `Napoleon` gleich 16 Computer braucht. Für den `Computerverlag` allerdings reichen läppische 16 Rechner sicher nicht aus...

Man beachte den komplizierten Zugriff auf die `Mhz`-Zahl des dritten Rechners der Firma `Diese`.

Zeiger auf Strukturen

Ein Zeiger auf eine Struktur ist fast dasselbe wie ein Zeiger auf eine normale Variable. Der Zugriff über den Zeiger auf einzelne Elemente ist dir anfangs vielleicht ein bisschen seltsam:

```
Firma *UltraFirma = &Diese;
(*UltraFirma).Mitarbeiter = 128;
```

Diese Schreibweise mit den Klammern ist wichtig, weil der Punkt eine höhere Priorität hat als der Stern. Würde man hier die Klammern weglassen, würde ein Zugriff auf das Feldelement `Mitarbeiter` erfolgen. Da der Zeiger jedoch auf eine Variable von `Firma` zeigt, und nicht auf dieses Element, gibt es einen Fehler.

Weil ein solcher Zugriff ziemlich oft in C++ verwendet wird, und weil diese Klammerkonstruktion etwas umständlich ist, hat man extra dafür einen Operator eingeführt. Er heißt ebenfalls Element-Auswahl-Operator und sieht so aus: `->`

```
UltraFirma->ComputerAnzahl++;
UltraFirma->Umsatz *= 1.3f;
```

Beispiel struct2.

Zeiger auf Elemente

Hier gibt es eigentlich nichts zu beachten, außer bei Zeigern auf Strukturen:

```
short *MitarbeiterNapoleon = &Napoleon.Mitarbeiter;
short *UmsatzUltraFirma = &(Ultrafirma->Umsatz);
```

Das war's dann erst mal mit Strukturen. Später wirst du Strukturen noch einmal im Zusammenhang mit objektorientierter Programmierung finden.

Unionen

In C++ gibt es noch einen anderen Konstrukt, der aber der Struktur ziemlich ähnlich ist: die Union. Es ist im Prinzip eine Struktur, allerdings wird bei einer Union jede Variable an der selben Stelle gespeichert - wird der Wert überschrieben, gibt es den alten Wert nicht mehr. Die Union ist vom Speicherplatz her gerade so groß wie ihre größte Variable. So sieht der Syntax aus:

```
union Name
{
   char Variable1;
   float X;
   float Y;
}CDU;
Name U1, Alpha;
```

Ähnlich der Struktur kannst du Instanzen einer Union direkt bei der Deklaration oder in einer extra Zeile erstellen.

Wenn etwa auf den Wert `X` zugegriffen wird, wird der Wert genommen, der gerade an der Speicherstelle steht. Zugriff auf einzelne Variablen erfolgt wieder mit dem Element-Auswahl-Operator, dem Punkt:

```
Alpha.X = 25.33;
cout << Y; //Der Wert von X wird eigentlich ausgegeben!!
```

Freie Unionen

Es ist auch möglich, den Namen der Union sowie jegliche Variablen dieses Typs wegzulassen. Dann brauchst du keine Elemente mehr auszuwählen:

```
union
{
   char Variable1;
   float X;
   float Y;
};
```

Zugreifen kann man dann wie auf normale Variablen:

```
Variable1 = 128;
cout << X; //128 wird ausgegeben (nicht als char sondern als float)
```

Beispiel union1.

Eine freie Union kann mit dem Schlüsselwort **static** initialisiert werden, wenn sie nicht gerade in einer Struktur ist. Dieses **static** bewirkt, dass eine Variable mit Null initialisiert wird. Man kann das aber umgehen, indem man diese Union in einer Struktur deklariert.

Eine Union macht dann Sinn, wenn es mehrere Optionen gibt, von denen aber nur eine ausgewählt sein soll.

Aufzählungstyp enum

Das ist so was wie eine Struktur aus lauter **int**-ähnlichen Variablen, die nach und nach sich um eins erhöhen. Um eine solche Aufzählung zu erstellen, gibt es das Schlüsselwort **enum** (von enumeration). Der Syntax dazu ist:

```
enum ZahlenVonNullBisFuenf { Null, Eins, Zwei, Drei, Vier, Fuenf };
```

Hier steht `Null` für `0`, `Eins` für `1` ... . Danach kann eine Variable dieses Aufzählungstyps deklariert werden. Diese Variable kann dann jeweils nur eine der Werte annehmen:

```
enum ZahlenVonNullBisFuenf AnzahlAepfel = Drei;
ZahlenVonNullBisFuenf AnzahlBirnen = Vier;
```

Das **enum** kann bei der Deklaration noch einmal vor dem Typ stehen (wie das bei C erforderlich war), muss (in C++) aber nicht!!

Man kann jetzt auch darauf zugreifen, sogar auch ohne eine Instanz zu benutzen:

```
if(AnzahlAepfel == AnzahlBirnen) Anweisungen;
if(AnzahlAepfel == Eins) Anweisungen;

int Zahl = 0;
Zahl = static_cast<int>(AnzahlAepfel);
AnzahlBirnen = static_cast<ZahlenVonNullBisFuenf>(Zahl);
```

enum-Elemente können direkt mit anderen Datentypen verglichen werden. Ihnen können aber nur mit einem Type-Cast ein anderer Datentyp zugewiesen werden, was andersrum nicht sein muss.

Beispiel enum1.

Soll die Aufzählung mit einer bestimmten Zahl beginnen, so kann man dem erstem Element auch gleich einen Wert zuweisen:

```
enum alpha {q=5, w, e, r, t, z, u }alpha1;
```

Hier hat q den Wert 5, w 6, e 7 und so weiter.

Namespaces

Ein **namespace**/Namensbereich kennst du schon, und zwar den **namespace** std. In diesem sind sämtliche Funktionen der C++-Standardbibliothek deklariert. Ein kurzes Statement dazu hab ich schon im ersten Kapitel gegeben. Hier wird das jetzt mal etwas genauer erklärt.

Namensbereiche kannst du dir wie Strukturen vorstellen. Sie sind dazu da, um mehrere Variablen oder Funktionen, die den gleichen Namen besitzen, beim Ändern und Lesen bzw. aufrufen richtig auszuwählen. Anders als bei Strukturen musst du allerdings keine Variable erstellen.

Zugriff auf einzelne Elemente erfolgt mittels des Bereichsoperators (::):

```
namespace Eins
{
   int A = 500;
   char B = 'f';
   float C = 13.24f;
}
namespace Zwei
{
   int B = 560;
   char A = 65;
   float C = 24.13f;
}
int B = 200;
short A = 100;

//Zugriff auf Elemente:
Eins::A = 400;
Zwei::C = 33.0f;
B = 644;
```

using namespace

Damit du nicht jedes Mal den Bereichsoperator mit Namen davor schreiben musst, gibt es das Schlüsselwort **using**. Dieses wird so verwendet:

```
using namespace Zwei;
```

Deswegen steht in jedem Beispielprojekt **using namespace** std;!

Falls es schon ein gleichnamiges Element außerhalb des Namespaces gibt, musst du eindeutig unterscheiden:

```
::B++;     //645    //Zugriff auf die globalere Variable
Zwei::B--; //559    //Zugriff ist weiterhin auch mit :: möglich
C *= 1.5;  //36.195 //Zugriff auf Zwei::C
```

Beispiel namespace1.

Mehrere gleiche Namen solltest du natürlich vermeiden. Falls es aber nötig ist, solltest du **namespace**s verwenden. **namespace**s können im Übrigen erweitert werden (im Sinne von Neudefinitionen).

Zusammenfassung

Mit neuem Wissen kannst du nun neue Probleme bewältigen! Einige Probleme eignen sich, um Felder einzusetzen. Andere wiederum sind dazu verpflichtet, mit Strukturen behandelt zu werden. Du wirst es selbst erfahren, wann das eine und wann das andere anzuwenden ist.

Strukturen sind die Vorreiter der objektorientierten Programmierung. Strukturen enthalten zunächst nur Variablen. Es gibt keine Begrenzungen bezüglich des Zugriffes auf die Elemente.

Auf Elemente einer Instanz einer Struktur greifst du mit dem Auswahloperator '`.`' zu. Wenn du das über den Umweg eines Zeigers tun willst, kannst du entweder das umständliche `(*Struktur).Element` oder das extra dafür erfundene `Struktur->Element` nehmen.

Unionen und Aufzählungstypen werden nur ziemlich selten angewandt. Sie sind eher für den Spezialfall gedacht. Namensbereiche wiederum sind recht nützlich, wenn es darum geht Verwechselungen zu vermeiden.

Workshop

Fragen

1.) Was kann eine Struktur alles beinhalten??
2.) Wo liegt der Unterschied zwischen Struktur und Union??
3.) Muss ein Aufzählungstyp bei Null beginnen??
4.) Bei welchen Fällen müsstest du den Operator `::` anwenden??
5.) Was ist hieran falsch??

```
struct A
{
   int A1;
   int A2;
   int *pA3;
};
```

6.) Und hier??

```
A m1;
A *pm1 = &m1;
m1.pA3 = pm1->A1;
```

```
(*m1).pA3 *= 5;

A &rm1 = pm1;
rm1.A1 = 3;
```

7.) Gibt es eine Alternative, um die Anweisung **using namespace** std; zu umgehen??
8.) Kann man mehrere **namespace**s gleichzeitig usen??

Links:

http://www.volkard.de/vcppkold/datenstrukturen.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop6_5.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop6_6.html
http://www.schornboeck.net/ckurs/struct.htm

Kapitel 11

Dynamische Speicherverwaltung

Nachdem ich Zeiger und dann komplexe Datentypen beschrieben habe, zeig ich dir nun die dynamische Speicherverwaltung. Die hat sowohl etwas mit Zeigern als auch mit Speicherbereichen zu tun, sodass ich kurz etwas Hintergrundwissen zu den verschiedenen Speicherbereichen vermitteln will:

Sämtliche Variablen, die während eines Programms benötigt werden, landen im Speicher. Dieser lässt sich dabei in 5 Teile unterteilen:

Codebereich:
Der Code des Programms und der darin enthaltenen Funktionen kommen in diesen Bereich. Fast jede Anweisung in C++ besteht aus mehreren Prozessoranweisungen (für den Prozessor bedeutet ein cout-Befehl eine Vielzahl von Einzelvorgängen). Die liegen nach dem compilern und linken in binärer Form vor und werden in den Codebereich kopiert.

Register:
Dieser Speicher liegt direkt im Prozessor. Zugriff darauf ist dadurch um einiges schneller als auf die restlichen Speicherbereiche. Ein Teil davon steht Variablen zur Verfügung, die als **`register`** deklariert wurden. Allerdings ist der Registerspeicher ziemlich begrenzt. Es kann mitunter vorkommen, dass der **`register`**-"Wunsch" nicht erfüllt werden kann.

Datenbereich:
Hier werden alle globalen (die globalsten aller Variablen) Variablen drin gespeichert. Die Speicherstellen werden nach Ablauf des Programms wieder freigegeben.

Stack:
Darin werden alle lokalen Variablen gespeichert. Der Speicher wird freigegeben, wenn eine Variable ihren Gültigkeitsbereich verlässt. Ein ständiges Kommen und Gehen.

Heap:
Dieser Speicher spielt eine große Rolle bei der dynamischen Speicherverwaltung. Dynamisch heißt, dass die z.B. Größe eines Feldes nicht von vornherein bekannt ist und sich während des Programms ändern kann (bei statischen Feldern gab entweder eine Konstante oder eine Variable die Größe eines Feldes EINMALIG an). Dieser reservierte Speicher wird allerdings nicht automatisch freigegeben; wenn also der entsprechende Befehl nicht kommt, ist ein Teil des Speichers erst wieder nach einem Neustart des Computers verfügbar.

Dynamische Speicherverwaltung ermöglicht es uns, Speicher so lange beizubehalten, bis er nicht mehr gebraucht wird und vielleicht einer anderen Variable zur Verfügung stehen soll. Dazu muss erst einmal Speicher auf dem Heap angefordert werden. Mit einem Zeiger kann dann auf den gesamten reservierten Speicherbereich zugegriffen werden. Sind wir fertig, löschen wir den Speicher aus der Speicherverwaltung.

Speicher reservieren

geht mit dem Schlüsselwort **`new`**. Es gibt eine Speicheradresse auf dem Heap zurück, die dann für einen Zeiger genutzt werden kann (nochmal zur Wiederholung: ein Zeiger ist

eine Variable, die eine Speicheradresse enthält; eine normale Variable kommt also gar nicht in Frage!).

Zunächst wird ein ganz normaler Zeiger deklariert. Diesem kannst du dann entweder nach der Deklaration eine Adresse mit **new** zuweisen oder, wie es häufiger gemacht wird, gleich bei der Deklaration die Adresse verpassen. Nach dem = steht das **new** und danach der Datentyp, dem auch der Zeiger entspringt.

Praktisch:

```
int *Zahl = new int;
```

Der jeweilige Datentyp muss nach **new**, damit der Compiler genügend Speicher reservieren kann. Das ganze kannst du aber auch (wie gesagt) so schreiben:

```
int *Zahl = 0;
Zahl = new int;
```

Hat der Zeiger einen Variablenzusatz, muss dieser auch nach das **new** geschrieben werden:

```
unsigned int *Zahl = new unsigned int;
```

Fehler

Es kann auch der Fehler auftreten, dass kein Speicher mehr auf dem Heap verfügbar ist. Für diesen Fall solltest du zumindest bei größeren Projekten überprüfen, ob der Speicher angelegt werden konnte (je größer ein Datentyp ist, desto wahrscheinlicher ist es, dass kein Speicher mehr verfügbar ist):

```
if(NULL == Zahl) cout << "Fehler bei Speicherreservierung";
```

Es ist nicht möglich, einen Zeiger oder eine Referenz auf dynamischen Speicher zu erzeugen, allerhöchstens ist ein Zeiger auf einen Zeiger möglich:

```
int **ZZ = new int*;
```

Dies erweist sich aber als unpraktisch, da eigentlich jeder Zeiger nur 4 Byte breit groß ist.

Den reservierten Speicher nutzen

Der Zugriff auf so eine dynamische Variable ist nicht anders als bei einem normalem Zeiger:

```
*Zahl = 25;
*Zahl += 33;
```

Du solltest den frisch reservierten Speicher am besten immer gleich einen Wert zuweisen. Auch auf dem Heap werden die Speicher so gelassen, wie sie verlassen werden. Einzig und allein die Speicherverwaltung bestimmt über Reserviertsein und Nichtreserviertsein.

Wie sieht es aus bei Feldern??

Es können auch gleich Felder deklariert werden. Die müssen dabei auch anders deklariert werden (man unterscheidet generell zwischen **new** und **new[]**). Die Feldgröße kommt in die eckigen Klammern:

```
double *f1 = new double[13];
char *f2 = new char[256];
```

Und wie wir das schon kennen, wird auf die Felder so zugegriffen:

```
f1[0] = 0.334423;
f1[12] = 12.315;

f2[0] = 'a';
cin >> f2;
f2[3] = '!';
```

Speziell für Strings solltest du ausreichend Speicher reservieren, damit `cin` nicht aus Versehen auf Speicher zugreift, den du gar nicht brauchen darfst.

Mehrdimensionale Felder

brauchen etwas mehr Aufwand. Anweisungen wie **char** `*f1 =` **new char**`[123][456];` ergeben Fehler. Also wie geht es denn, wenn einmal ein mehrdimensionales Feld von Nöten sein sollte??

Es gibt sozusagen soviele Schritte wie es Dimensionen geben soll. Wir brauchen einen Zeiger auf einen Zeiger, um erst einmal ein zweidimensionales Array aufzubauen. Der bekommt ein Feld von Zeigern dynamisch angelegt:

```
int **f2 = new int*[13];

for(int i = 0; i < 13; i++) //<- 13 beachten
{
   f2[i] = new int[256];
}
```

Jetzt haben wir ein Feld mit 13 Elementen in der ersten Dimension und 256 Elementen in der zweiten. Bloss nicht die Indexzahlen tauschen!!! Korrekter Zugriff auf ein Element:

```
f2[0][0] = 0;
f2[3][3] = 9;
f2[12][255] = 12255; //letztes Element
```

Umso kritischer wird es bei noch mehr Dimensionen. Hier ein dreidimensionales Monstrum:

```
int ***f3 = new int**[60];

for(int i = 0; i < 60; i++)
{
   f3[i] = new int*[13];
   for(int i2 = 0; i2 < 13; i2++)
   {
      f3[i][i2] = new int[256];
   }
```

```
}

f3[0][0][0] = 4+3;
f3[59][12][255] = 5912255; //letztes Element
```

Gleich wirst du sehen, wie diese beiden Felder wieder aus dem Speicher entfernt werden.

Speicher wieder Freigeben

Mit **delete** und **delete[]** kannst du durch **new** bzw. durch **new[]** reservierten Speicher wieder freigeben, um ihn anderen Programmen bzw. Variablen deines Programms zur Verfügung zu stellen. **delete** setzt du bei Einzelvariablen und **delete[]** bei Feldern ein. Die bereits reservierten Elemente könnten wir so löschen:

```
delete Zahl;
delete[] f1;
delete[] f2;
```

Der Speicher wird vollkommen gelöscht und die Daten gehen verloren (bei einem mit **new[]** reservierten Feld wird durch **delete** nur das erste Element freigegeben - deshalb **delete[]**). Mehrdimensionale Felder müssen wieder extra über Schleifen aus dem Speicher geholt werden:

```
for(int i = 0; i < 13; i++)
{
   delete[] f2[i];
}
delete[] f2;

for(int i = 0; i < 60; i++)
{
   for(int i2 = 0; i < 13; i++)
   {
      delete[] f3[i][i2];
   }
   delete[] f3[i];
}
delete[] f3;
```

Wenn du Daten behalten willst, musst du davon Kopien machen. Um die selben Daten in ein größeres Feld zu schreiben, musst du sie kopieren, das Feld löschen und mit mehr Elementen neu erschaffen - alles mit demselben Zeiger.

Memory Leaks

Falls du mehrmals hintereinander mit **new** neue Speicheradressen für eine Variable holst, den Speicher aber nicht wieder freigibst, kommt es zu sogenannten Memory Leaks (Speicherleck):

```
Zahl = new int[1024]; //noch ist die Welt in Ordnung!!

Zahl = new int; //die erste Speicheradresse geht verloren und 4096
Byte Speicher sind verschwendet

Zahl = new int[26]; //weitere 4 Byte gehen verloren!!
delete Zahl; //4 Byte werden wieder frei; der Rest von 100 Byte
bleibt aber reserviert!!
```

Memory Leaks können mehr oder weniger offensichtlich passieren. Das obere Stückchen Code gehört wohl eher zu den offensichtlichen Varianten.

Beispiel new1.

Zusammenfassung

kann ich mir sparen,

Workshop

wohl auch!

Die letzten drei Kapitel waren ziemlich heftig. Es erfordert viel Übung, um diese Dinge drauf zu haben. Nimm dir die Zeit (ich kann es gar nicht oft genug betonen!) und programmiere, was das Zeug hält!

Kommende Kapitel erfordern noch mehr logisches Denken und Ausdauer - manche mehr manche weniger. Sicherlich wirst du nicht jede Möglichkeit nutzen, die hier in diesem Tutorial beschrieben ist, aber fast alle Konstrukte wirst du in den Standard-Headern vorfinden können, und zudem, je mehr du (mit großem Sprachumfang) programmierst, desto leichter fällt dir, Code von anderen Leuten zu lesen - das ist besonders in einem Team wichtig!!! Du solltest jegliche Trainingsmöglichkeiten nutzen.

GDO kann mit diesem ganzem (naja, Strukturen und Felder) Zeug teilweise technisch verbessert werden.

Auch hier solltest du wieder versuchen, das Programm zu verstehen. Im Vergleich zum vorigem habe ich es technisch sowie storytechnisch verbessert. Der Umfang ist auf etwa 700 Zeilen gewachsen, nicht oft wegen der Technik.

Links:

http://www.volkard.de/vcppkold/new_und_delete.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop8.html
http://www.schornboeck.net/ckurs/alloc.htm

Kapitel 12

In diesem Kapitel soll es um verschiedene Programmiertechniken gehen, auch Programmieransätze. Daraus wird dann die Modularisierung hervorstechen (ich verspreche ein weniger anstrengenden Kapitel als die vorigen).

Programmieransätze

Bis zum Kapitel 5 war alles algorithmisch orientiert. Es handelt sich um Beispiele, die jeweils nur Abfolgen von Befehlen sind. Dazu zählt auch die Möglichkeit, Code durch Schleifen wiederholen zu lassen (iterativ). Ein kleines Beispiel:

```
int main(void)
{
   int a = 0;
   int b = 5;

   a = 33;
   cout << "a : " << a;
   a = 23;
   cout << "a hat den Wert " << a;
   a = a * 4 - 19;
   cout << "a hat den Wert " << a;

   for(a = 1; a < 6; a++)
   {
      b *= a;
      cout << "b : " << b;
   }

   cin >> i;

   return 0;
}
```

Prozedurale Programmierung

Ab Kapitel 6 gab es dann die Funktionen, die für den prozeduralen Programmieransatz wichtig sind. Der Code ist somit universeller geworden und du musst nur eine Funktion aufrufen, um eine ganze Reihe von Befehlen ausführen zu können. Das macht Sinn, wenn diese Reihe von Befehlen zwei oder mehrere Male ausgeführt werden soll.

Wenn du zwei Blöcke hast, die sich nur geringfügig voneinander unterscheiden, solltest du sie zu einer Funktion vereinigen und mit Parametern einzelne Fälle bestimmen. Ein recht simples Beispiel:

```
int main(void)
{
   int i = 0;

   i = 33;
   cout << "i hat den Wert " << i;
   i = i * 3 + 9;
   cout << "i hat den Wert " << i;

   i = 39;
   cout << "i hat den Wert " << i;
```

```
   i = i * 4 - 19;
   cout << "i hat den Wert " << i;

   cin >> i;

   return 0;
}
```

Hier siehst du, dass sich die beiden Blöcke ähnlich verhalten, deshalb kann man das ganze ein bisschen optimieren:

```
int FunktionFuerI(int Z1, int Z2, int Z3, char Name[256] = 0);

int main(void)
{
   int i = 0;
   i = FunktionFuerI(33, 3, 9, "i");
   i = FunktionFuerI(39, 4, -19, "i");
}

int FunktionFuerI(int Z1, int Z2, int Z3, char Name[256] = 0)
{
   cout << Name << " hat den Wert " << Z1;
   Z1 = Z1 * Z2 + Z3;
   cout << Name << " hat den Wert " << Z1;

   return Z1;
}
```

Auch wenn hier die Funktion nur zweimal aufgerufen wird, lohnt sich der Eingriff, weil eine Änderung (etwa / anstatt * bei der Zuweisung) nur einmal gemacht werden muss. Zudem ist es jetzt sehr leicht möglich, noch hundert weitere Aufrufe zu machen, wobei der Schreibaufwand verringert wird (selbiges passiert auch mit der Größe der Quellcodedatei und der daraus resultierenden Programmdatei). Und übersichtlicher wird das Programm auch noch.

Soweit die prozedurale Programmierung.

Modulare Programmierung

Man kann auch noch einen Schritt weitergehen, und ein Projekt in mehrere Module (auch Übersetzungseinheiten genannt) aufspalten. So was nennt man dann modularisierten Programmieransatz. Dabei sollten möglichst logisch zusammenhängende Teile in jeweils eine Datei. In deinen Projekten kannst du entweder Header-Dateien oder *.cpp-Dateien erzeugen.

Sieh dir zunächst das **Beispiel modul1** an.

In diesem Projekt befinden sich zwei *.cpp-Dateien, eine Headerdatei *.h und noch eine weitere Headerdatei *.hpp. Es gibt keinen Unterschied zwischen *.cpp und *.cc, *.c++, *.cp oder *.cxx-Dateien, man kann sogar *.c-Dateien in ein C++ Projekt mit einbeziehen. Jedoch solltest du immer nur cpp-Dateien (oder Dateien mit den eben genannten Dateiendungen) einfügen, weil das darauf hinweist, dass es sich um ein C++-Projekt handelt, während *.c-Dateien eigentlich nur bei C-Projekten zum Einsatz kommen. Bei den Header-Dateien gibt es zu sagen, dass es die mit dem h am Ende immer bei C-Projekten gab. Da sich die h- und hpp-Dateien jedoch nicht wirklich unterscheiden, kann man sie auch bei C++-Projekten benutzen. Du solltest also immer *.h- und *.cpp-Dateien verwenden.

In eine der cpp-Dateien muss die `main`-Funktion. In eine h-Datei können die Prototypen sämtlicher Funktionen sowie Deklaration (sowie Initialisierungen) einiger Variablen und Definitionen mit `#define`. In eine andere cpp-Datei können die Definitionen der Funktionen aus der h-Datei. Wenn man das so ordnet, kann man eine 1000-Zeilen-Datei in mehrere 100-Zeiler aufteilen, was deutlich zur Übersichtlichkeit beiträgt.

Die h-Dateien müssen in die cpp-Dateien, in denen etwas davon verwendet wird, eingebunden werden. Das passiert auch hier wieder mit `#include`, diesmal muss die h-Datei allerdings in Anführungszeichen gesetzt werden, damit der Compiler weiß, dass die Datei sich nicht im Standard-Include-Verzeichniss, sondern im Projektverzeichniss befindet.

static

Da eine Variable, definiert in einer Datei, in einer anderen Datei verfügbar ist, muss sie dort nicht extra noch einmal neu deklariert werden. Manchmal ist es aber nötig, dass zwei Variablen mit dem selben Namen in zwei verschiedenen Dateien unterschiedlich genutzt werden.

Bei der freien Union hab ich geschrieben, dass man **`static`** dazu nutzen kann, um eine solche Union global zu deklarieren. Dem Schlüsselwort kommt neben der Initialisierung einer Variablen mit dem Wert `0` aber noch eine weitere (wichtigere) Bedeutung zu: es bewirkt, dass eine Variable oder Funktion nur innerhalb einer Übersetzungseinheit verfügbar ist. Zwei Variablen mit gleichen Namen sind also in zwei unterschiedlichen Files vollkommen unabhängig voneinander.

extern

Mit dem Schlüsselwort **`extern`** kann man bestimmen, dass eine Variable auch außerhalb seines Gültigkeitsbereiches verfügbar ist, und sie sich deswegen wie eine Referenz auf die Variable verhält.

Es ist mit **`extern`** möglich, dass eine globale Variable auch nach einer lokalen Neudefinition verfügbar ist:

```
#include <iostream>

using namespace std;

int AAA = 25;

int main(void)
{
   cout << AAA; // 25
   extern int AAA = 18889 ; //ist Referenz zur globalen Variable
AAA
   cout << AAA ; // 18889
   {
      int AAA = 128; //ist eine lokale Variable
      cout << AAA; // 128
      cout << ::AAA; // 18889 (globalere Variable)
   }

   cin >> AAA;

   return 0;
}
```

Im Zusammenhang mit Modulen gibt es noch einen weiteren Verwendungszweck, und zwar eine Variable in zwei verschiedenen Dateien:

```
//Datei1.cpp:
int HUHN = 25;           //in Datei 1 und 2 verfügbar als eine
Variable
int HAHN = 161;          //in Datei 1 und 2 verfügbar als zwei
Variablen
static int KUECKEN = 121;//in Datei 1 und 2 verfügbar als zwei
Variablen

//Datei2.cpp:
extern int HUHN = 31;
static int HAHN = 9;
static int KUECKEN = -33;
```

Beispiel modul2.

Es ist aber nicht möglich (und auch nicht sinnvoll), beide Schlüsselwörter zugleich anzuwenden (`static extern int PPP;`).

Zusammenfassung

C++-Compiler bieten die Möglichkeit, mehrere Quellcodedateien zu einer Exe zu übersetzen. Erst mehrere Codedateien machen ein richtiges Projekt aus. Dev-C++ erstellt aus den Informationen in den *.dev-Dateien die richtigen Parameter, um alle im Projekt enthaltenen Sources zu einem Programm zu verbinden. Wenn du mal in den "Compiler Log" schaust, werden dich die an den Compiler übergebenen Parameter verwirren. Dev-C++ macht das automatisch und wir können uns auf das Coden konzentrieren. Ohne diese IDE müssten wir den Compiler mit diesen Angaben selbst versorgen.

In cpp-Dateien gehören generell Definitionen (Funktionen, z.B. die `main`-Funktion). Header Dateien, die mit h enden, enthalten meist Deklarationen.

Workshop

Fragen

1.) Wieso unterteilt man überhaupt in Header- und cpp-Dateien??
2.) Lohnt es sich, jedes Projekt, auch das kleinste, so zu unterteilen??
3.) Wie könnte man das eine Projekt von anderen Projekten abgrenzen??

Programme

1.) Such dir 3 bis 10 Projekte aus früheren Kapiteln aus und teile ihre Quellcodedateien in mehrere Files auf!

Die Modularisierung lässt sich gut auf **GDO** anwenden. Hier wird jedoch **`static`** nicht verwendet.

Links:

http://www.volkard.de/vcppkold/uebersetzungseinheiten.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop2_2.html
http://www.schornboeck.net/ckurs/module.htm

Kapitel 13

Dieses Mal geht es um Anweisungen im Code, die in Richtung Projektmanagement zielen. Es kommt also ganz gelegen nach dem Kapitel mit den Programmieransätzen, speziell der Modularisierung.

Zwei Anweisungen kennst du ja schon - `#include` und `#define`!

Präprozessordirektiven

Das sind Anweisungen, die nicht direkt ins Programm eingreifen. Sie dienen dem Projekt und sind für dich eigentlich ganz praktisch. Der Präprozessor ist einer der ersten Sachen, die beim Compile-Vorgang durchgeführt wird. Er wandelt den Quellcode in gewissem Maße um.

Alle Präprozessordirektiven beginnen mit einer Raute '`#`' und enden im Zeilenende. Ein Semikolon ist demnach nicht nötig. Allerdings kannst du einen langen Inhalt auf mehrere Zeilen aufspalten, indem du am Ende der jeweiligen Zeile ein '`\`' setzt und dann gleich mit der nächsten Zeile beginnst.

Zunächst zeige ich dir noch mal die zwei Direktiven `#define` und `#include`:

#define

Damit kannst du Konstanten und Makros definieren (andere Nennweise Makros und funktionsähnliche Makros). Die gelten dann im jeweiligen Modul, wo du sie definiert hast, außer du bindest das Modul in ein anderes Modul mit ein.

Konstanten kannst du so definieren:

```
#define NAME "Wert Text Befehl"
```

So, nun ist die Konstante `NAME` mit dem Ausdruck `"Wert Text Befehl"` definiert. Das erste Wort nach `#define` ist der Name der Konstanten und alles, was danach kommt, ist der Ausdruck, für den `NAME` steht. Überall, wo du die Konstante hinschreibst, wird deren Name durch den definierten Ausdruck ersetzt.

Also wird folgendes:

```
printf("%s", NAME);
```

beim compilern zu:

```
printf("%s", "Wert Text Befehl");
```

Makros gehen so:

```
#define MAKRO(a, b, c) cout << a << b << c;
```

Ein Makro erkennt man an seiner Parameterliste. Diese muss dem Namen ohne Leerzeichen folgen. Das, was dem Makro übergeben wird, wird auch gleich in den Ausdruck übertragen.

So würde:

```
MAKRO(23, "Hallo !!?", '?');
```

beim compilern zu:

```
cout << 23 << "Hallo !!?" << '?';
```

Beispiel define1.

Für weitere Informationen kannst du ja noch mal zu **Kapitel 3 Abschnitt Konstanten** gehen. Wie dort schon erwähnt, sollten Konstanten und Makros grundsätzlich großgeschrieben werden. Im nächsten Kapitel wird es noch mehr um Makros gehen und du wirst auch mehr Beispiele dazu finden.

#include

Mit diesem Befehl kannst du Übersetzungseinheiten (Module) in eine andere Übersetzungseinheiten einbinden. Dadurch kannst du dann auf die darin verwendeten Funktionen und Konstanten (und Makros) zugreifen.

Eine Headerdatei, die sich im Standardverzeichnis des Compilers befindet, bindest du mit eckigen Klammern ein:

```
#include <iostream>
```

Ist die Datei jedoch woanders, musst du das in Anführungszeichen angeben:

```
#include "MyHeader.h"
```

Befindet sich der Header in einem Unterverzeichnis, kannst du das mit '\' klarmachen:

```
#include <std\bastring>
#include "Ordner\MyHeader"
```

Beispiele gibt's genug.

#undef

Wie der Name schon andeutet, löscht `#undef` eine vorher mit `#define` definierte Konstante (oder Makro). `#undef` muss nur den Namen der Konstanten oder des Makros als Parameter bekommen, und schon kannst du den freigewordenen Namen für was anderes gebrauchen.

```
#define HANS 20
//...
#undef HANS
#define HANS 50
```

Bis zum `#undef HANS` hat `HANS` den Wert `20` , danach jedoch `50`. Kommt eine Konstante im Code nach einem solchen `#undef`, gibt der Compiler das als Fehler aus.

#if , #elif , #else , #endif , #ifdef und #ifndef

Auch beim Präprozessor kannst du Fallunterscheidungen machen lassen. Das Resultat ist die sogenannte "Selektive Compilierung". Das heißt, dass ein Teil des Codes nur dann compilert wird, wenn eine bestimmte Bedingung erfüllt wird.

Mit dem `#if` leitest du einen **`if`**-Zweig ein:

```
#if HANS == 25
```

Das entspricht etwa dem hier:

```
if (HANS == 25)
{
```

Da fehlt tatsächlich noch eine schließende Klammer. Diese Rolle übernimmt das `#endif`. Normalerweise schreibt man danach in ein Kommentar, wie die Bedingung für diesen Zweig war.

```
#endif    //HANS == 25
```

Nun gibt es dazu noch die beiden Direktiven `#elif` und `#else`. `#elif` übernimmt die Aufgabe von **`else if`** und `#else` steht natürlich für **`else`**. !!!Nur `#if` ist mit einem `#endif` abzuschließen!!! Hier mal eine komplette Struktur:

```
#if HANS < 25
#undef HANS
#define HANS 25

#elif HANS > 75
#undef HANS
#define HANS 75

#else
#undef HANS
#define HANS 50
#endif    //HANS < 25
```

Zwei besondere Direktiven sind `#ifdef` und `#ifndef`. Ein `#ifdef` prüft, ob sein Parameter definiert worden ist. Wenn ja, wird der nachfolgende Code compiliert. `#ifndef` tut da genau das Gegenteil: Wenn die Konstante NICHT definiert wurde, wird der nachfolgende Code compiliert.

Beide Direktiven können jeweils nur einen Parameter annehmen und müssen mit `#endif` beendigt werden. Außerdem können sie nur Konstanten, die mit `#define` definiert wurden, annehmen. Der Wert einer solchen Konstanten (wenn sie überhaupt einen Wert hat) ist `#ifndef` und `#ifdef` egal.

```
#ifdef HANS
#undef HANS
#define HANS NEUERWERT
#endif    //ifdef HANS
```

Da diese jedoch immer nur einen Parameter überprüfen können, hat man für `#if` noch einen Zusatz geschaffen: `defined`. Mit einer geeigneten Konstruktion kannst du mehrere Konstanten auf ihr Dasein prüfen:

```
#if (defined HANS) && (!defined BERT) || (defined CRAB)
```

Nur mit `#ifdef` und `#ifndef` geschrieben würde das so aussehen:

```
#ifdef HANS
#ifndef BERT
//...
#endif    //ifndef BERT
#endif    //ifdef HANS

#ifdef CRAB
//...
#endif    //ifdef CRAB
```

Der Include-Wächter

Mit einer speziellen Konstruktion kannst du das mehrfache Einbinden eines Headers verhindern. Diese Idee wird auch in der Standardbibliothek angewandt.

In jedem Header wird eine Konstante definiert, die etwa dem Namen des Headers entspricht. Vor dieser Definition wird geprüft, ob die Konstante schon definiert wurde. Wenn nicht, dann wird sie definiert und der eigentliche Inhalt des Headers wird compiliert. Wird der Header anschließend noch mal eingebunden, wird der Inhalt aber nicht noch einmal compiliert. Dadurch entstehen keine Neudefinitionen.

Beispiel include1.

#error

Wenn der Compiler auf eine solche Direktive stößt, gibt er das, was nach `#error` steht, als Fehler aus. Das ist z.B. hilfreich, um zu prüfen, ob ein bestimmter Header (mit seinem Include-Wächter) eingebunden ist.

```
#ifndef _MYHEADER_H_
#error MyHeader.h fehlt!!!
#endif //ifndef _MYHEADER_H_
```

Weitere Direktiven

Zu den oben genannten gibt es noch `#line` und `#pragma`. !!!`#line` ist jedoch nicht für Dev-C++ (mit Compiler gcc) verfügbar (es ist eine Erfindung des MS Visual C++-Compilers), wird aber auch nicht als Fehler angesehen!!! `#pragma` kann Einstellungen für den Compiler vornehmen. Für `#pragma` gibt es eine Reihe von Parametern, die ein eigenes Tutorial füllen würden. Gehen wir einfach davon aus, dass wir an solche Spezialfälle nicht rankommen, dass ein `#pragma` nötig wäre.

Zusammenfassung

Präprozessordirektiven dienen wie gesagt dem Projekt. Mit selektiver Compilierung ist es möglich, Code auf einem System zu programmieren und den Code auf einem anderen System an die geänderten Bedingungen des Systems anzupassen.

Zum Beispiel schreibt ein Programmierer die Textverarbeitung Word 2007 für die x86-Plattform (moderne Windows- oder Linux-PCs). Weil mobile Minicomputer grad im Trend sind, passt besagter Programmierer den Code auf diese Plattform an. Für den Macintosh bastelt er auch gleich eine Version. Auf jeder dieser Plattformen kann man mit C++ programmieren. Die Compiler fallen dementsprechend unterschiedlich aus. Die gleichen Sourcecodes werden dann auf den verschiedenen Plattformen compilert.

Vielleicht wirst du dieser Programmierer eines Tages sein, vielleicht auch nicht. Egal. (Wenn ja, wirst du schmunzelnd auf diesen Kapitel zurückblicken.)

Zunächst erst einmal wichtiger ist der Nutzen, der sich mit dem `#include` ergibt. Du wirst in jedem Header der Standardbibliothek den Include-Wächter entdecken können. In künftigen Beispielprojekten wirst du diese Technik auch finden.

Workshop

Fragen

1.) Gab es die Anweisungen schon in C, oder wurden die erst mit C++ erfunden??
2.) Was passiert, wenn man einen Header doppelt einbindet, der keinen Include-Wächter hat??
3.) Können diese Anweisungen überall stehen, auch in Funktionen drin??
4.) Wie lauten die Verzweigungen für:

Wenn `A 100` ist und `B 300`??
den Include-Wächter von **iostream**??
Wenn `X` definiert ist und `Y` größer als `300` aber kleiner als `400`??

5.) Finde die Fehler:

```
#define HANSHEADER
#define FISCHER 3902

#ifdef HANSHEADER
#define HANSHAEDER 345

#ifdef FISCHER
#define FISCHER 40390;

#if(defined FISCHER || HANSHEADER)
#define HANSHEADER_AND_FISCHER_DEFINED
#endif; //#if(defined FISCHER || HANSHEADER)
```

Programme

1.) Rüste die Beispielprojekte vom Modularisierungs-Kapitel mit dem grade gelernten aus!!
2.) Rüste alle GDO-Versionen mit Modularisierung und Include-Wächter aus!!
3.) Dasselbe nochmal mit den Projekten, die du selbst mit Modulen programmiert hast!!

Links:

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop2_2.html
http://www.schornboeck.net/ckurs/preprocessor.htm

Kapitel 14

In diesem Kapitel will ich dir - wie der Titel schon sagt - einige nützliche und sinnvolle Sachen zeigen, die den "C++ - Alltag" etwas erleichtern bzw. verbessern. Du wirst die C-Funktion `printf()` kennen lernen - das Ausgabe-Flaggschiff der alten (pardon, Vorgänger-)Sprache. Dann werde ich im Abschnitt Makros genauer auf die Fähigkeiten der `#define`-Direktive eingehen. Beides zusammen führt dich dann zu Funktionen mit variabler Argumentzahl. Am Ende lernen wir noch eine wichtige Funktion kennen: `rand()`, um endlich einmal ein bisschen Zufälligkeit in GDO zu bringen!

Im nächsten nützlichen Kapitel geht es dann nützlich weiter mit nützlichen Dateibehandlungs-Routinen, ebenfalls aus C.

Ausgabe mit printf()

Alle Variablen und Texte, die bisher auf dem Bildschirm landeten, wurden mithilfe von `cout` ausgegeben. Neben diesem Ausgabeobjekt gibt es aber auch noch etwas anderes zur Ausgabe auf den Bildschirm, und zwar die Funktion `printf()`. Damit lassen sich Daten mit ähnlichen Möglichkeiten wie mit `cout` ausgeben. Sie stammt aus der vorgehenden Sprache C und ist ein bisschen angestaubt. Aber du wirst merken, dass `printf()` ein bisschen schneller ist und auch die Exe nicht so aufbläht wie die iostream-Klassenbibliotheken (zu denen ich später komme).

Sie ist in **cstdio** definiert und hat folgenden Aufbau:

```
int printf(const char *format, ...);
```

Der Funktion wird zuerst ein Argument in Form eines Strings übergeben, danach können beliebig viele Argumente, egal welchen Datentyps, übergeben werden. Diese optionale Anzahl an Argumenten soll uns weiter unten beschäftigen. Damit `printf()` alles gut formatiert ausgeben kann, müssen dem String verschiedene Zeichenketten eingefügt sein. Diese Zeichenketten sind so aufgebaut:

```
%<Ausrichtung><Breite><.Nachkommastellen>Typ
```

Damit `printf()` weiß, dass eine Variable oder Konstante ausgegeben werden soll, musst du an geeigneter Stelle ein Prozentzeichen schreiben. Die drei folgenden Wörter in den eckigen Klammern sind optional. Bei `Ausrichtung` kannst du folgende Zeichen hinschreiben:

`0` - Zahl wird von links mit Nullen gefüllt
`-` - Ausgabe erfolgt linksbündig
`+` - auszugebenden Zahlen wird ein Vorzeichen vorangestellt (wenn eine Zahl positiv ist, so wird das Plus davorgestellt)

Mit `Breite` kannst du bestimmen, wie viele Zeichen für eine Ausgabe vorgesehen werden. Ganzzahlen werden allerdings ganz ausgegeben, auch wenn ihnen kein ausreichender `Breite`-Wert zugewiesen wird. Wenn du die Angabe weglässt, gibt `printf()` den Wert mit ausreichender Breite aus.

Mit `Nachkommastellen` kannst du bestimmen, wie viele Stellen nach dem Komma bei einer Fließkommazahl ausgegeben werden sollen. Vor den Wert musst du einen Punkt setzen, damit `printf()` bescheid weiß.

Damit eine Variable auch in der richtigen Form ausgegeben wird, musst du noch den Typ angeben. Den kannst du der folgenden Tabelle entnehmen:

Zeichen	Datentyp	Ausgabeformat
`c`	**`char`**	ein Zeichen
`s`	**`char*`** oder **`char[]`**	Zeichenkette/String
`d` oder `i`	**`int`**, **`long`** und **`short`**	Dezimalwert
`u`	**`unsigned int`** und **`short`**	vorzeichenloser Dezimalwert
`o`	**`int`**, **`long`** und **`short`**	Oktalwert
`X` oder `x`	**`int`**, **`long`** und **`short`**	Hexadezimalwert
`f`	**`float`** und **`double`**	Fließkommawert
`E` oder `e`	**`float`** und **`double`**	Fließkommawert in Exponentialschreibweise

Beispiel printf1.

Als Rückgabewert gibt `printf()` ein **`int`** zurück, deren Wert der Anzahl der ausgegebenen Zeichen entspricht.

Um uns mal vollkommen von der **iostream** Bibliothek zu trennen, steht am Ende der `main`-Funktion `getchar()`. Diese Funktion ist ebenfalls in **cstdio** definiert und dient dazu, eine Eingabe vom Benutzer zu erwarten, wie es zuvor immer mit `cin >> Variable;` gemacht wurde. Beachte die Größe der Anwendung, die beträchlich unter der einer Anwendung mit **iostream** liegt.

Makros

Jaja, Makros wurden schon im vorigen Kapitel durchgekaut. Jetzt geht das aber gleich ein bisschen ausführlicher los.

Ein Makro ist wie eine sehr klein gehaltene Funktion mit ein oder zwei Anweisungen. Aber so etwas wie eine Deklaration gibt es nicht - das ist erstens gar nicht nötig und zweitens mit `#define` überhaupt nicht möglich. So kommt man gleich zur Definition:

```
#define MAKRO(Parameter) Makroinhalt
```

Wichtig ist, dass zwischen der Parameterliste (wo `Parameter` drin steht) kein Leerzeichen steht. Sonst würde die Parameterliste mit zum Makroinhalt gehören. Es dürfen auch keine Datentypen vor die Parameternamen kommen - die Parameter dienen schließlich nur dazu, um im Makrorumpf durch die Argumente ersetzt zu werden.

Jetzt kommt, was man bei Funktionen den Funktionsrumpf nennt, und zwar eine gewünschte Kette von Befehlen. Dabei gibt es zwei Möglichkeiten, was die Form angeht:

Als Einzelanweisung

```
#define ADD(Zahl1, Zahl2) (Zahl1 = Zahl2 = Zahl1 + Zahl2);
int Argument1 = 10;
int Argument2 = 680;

//Aufruf:
ADD(Argument1, Argument2)
```

Nutzt ein Makro noch weitere Variablen oder Konstanten, die nicht als Argumente übergeben werden, müssen diese jedoch auch in dem Bereich, wo das Makro aufgerufen wird, zur Verfügung stehen. Deswegen ist es unbedingt ratsam, alle verwendeten Variablen/Konstanten als Parameter zu übergeben.

Wenn ein Makro aufgerufen wird, wird eigentlich nur der Code, der mit einem bestimmten Namen (hier ADD())aufzurufen ist, an die Stelle des Aufrufs gesetzt und dieser ersetzt dann den Ausdruck des Aufrufs. Die Parameternamen werden dabei durch die Namen der übergebenen Argumente ersetzt, wenn man also oben kein Makro verwenden würde, würde man folgenden schreiben:

```
int Argument1 = 10;
int Argument2 = 680;

//Aufruf:
(Argument1 = Argument2 = Argument1 + Argument2);

/* Die Klammern haben nichts zu sagen, außer dass sie angeben,
welche von eventuell vielen Operationen zuerst ausgeführt werden
sollen. Man kann auch schreiben:*/

// Argument1 = Argument2 = Argument1 + Argument2;
```

Beispiel makro1.

Als Anweisungsblock

```
#define DELETE(*Pointer) {delete Pointer; Pointer = 0;}
char *String = new char[256] ;
//Aufruf:
DELETE(String)
```

Hier wird eigentlich ein Block geschrieben, deswegen gehen auch mehrere Anweisungen. Formal schöner wäre die Formatierung, wie sie bei Funktionen üblich ist:

```
#define DELETE(*Pointer) \
{                        \
    delete Pointer;      \
    Pointer = 0;         \
}

char *String = new char[256] ;
//Aufruf:
DELETE(String)
```

Die Backslashs solltest du wie hier auf einheitliche Höhe bringen, sonst könnte noch einer durcheinanderkommen. Egal ob die zweite Variante schöner aussieht, in der Praxis ist das einzeilige Makro dann doch meist besser zu überschauen.

Beispiel makro2.

Operatoren für #define

Für `#define` gibt es noch extra 2 Operatoren, die es sonst nirgendwo gibt. Dann noch der ?:-Operator, den du in Makros so manches mal antreffen wirst.

Zeichenfolgenoperator

Er wandelt ein Makroparameter in einen String um. Du musst ihn vor den Parameter schreiben.

Beispiel define2. (Dieser Abschnitt stand ursprünglich woanders, also nicht wundern über die Projekt-Namen)

Vereinigungsoperator

Damit kannst du zwei Makroparameter zu einem Namen verbinden:

```
#define CONNECTOMAT(a, b) a##b
```

Beispiel define3.

Ab und zu sind diese beiden ganz nützlich, wenn auch nicht der beste Programmierstil.

Der ternäre ?:-Operator

Dieser Operator, den ich dir schon ziemlich am Anfang (Kapitel 4 wars wohl) erklärt habe, befindet sich sehr oft in Makros. Deswegen hier noch eine kleine Abhandlung.

```
#define IsEqual(a, b) a==b?a:b
#define IsBigger(a, b) a>b?a:b

int Z1 = 100;
int Z2 = 200;
int Z3 = 200;
int WhatIs = 0;

//Aufruf:
WhatIs = IsEqual(Z2, Z1); //WhatIs ist 100
WhatIs = IsEqual(Z2, Z3); //WhatIs ist 200
WhatIs = IsBigger(Z2, Z1); //WhatIs ist 200
WhatIs = IsBigger(Z3, Z1); //WhatIs ist 200
```

Zuerst steht die Bedingung, dann ein Fragezeichen und dann kommt der Ausdruck, die benutzt werden soll, wenn die Bedingung zutrifft. Dann ein Doppelpunkt mit dem Ausdruck dahinter, die genutzt werden soll, wenn die Bedingung nicht zutrifft. Benutzt werden heißt hier, dass der gesamte Ausdruck durch einen der beiden Ausdrücke, die durch die **if-else**-Anweisung bestimmt werden, ersetzt wird. Somit wird bei dem erstem Aufruf `Z1` `WhatIs` zugewiesen, beim zweitem wird `Z2`, beim drittem `Z2` und beim viertem `Z3`.

Beachte, dass bei keinem Makroaufruf ein Semikolon nötig ist, sehr wohl aber bei einer Zuweisung wie etwa `WhatIs = IsEqual(Z2, Z1);`.

Beispiel makro3.

Manchmal wird eine #define-Konstante als Makro und ein Makro von oben als funktionsähnliches Makro bezeichnet.

Funktionen mit variabler Parameteranzahl

Die printf()-Funktion hat wie oben erwähnt eine variable Anzahl von Argumenten. Um das auch in eigenen Projekten bewerkstelligen zu können, gibt es eine Reihe von Makros in der Header-Datei **cstdarg**. Diese sind:

```
#define va_start(ap, pN) ((ap) = ((va_list)
__builtin_next_arg(pN)))

#define va_arg(ap, t) (((ap) = (ap) + __va_argsiz(t)), *((t*)
(void*) ((ap) - __va_argsiz(t))))

#define va_end(ap) ((void)0)
```

ap ist jeweils ein Parameter vom Typ va_list (va_list ist ein als **char*** definierter Typ) , der die Liste der Parameter der Funktion mit der variablen Parameterzahl enthält. pN ist das Argument, das vor dem ersten optionalen Parameter in der Parameterliste steht. Wie pN mit __builtin_next_arg in Zusammenhang tritt, brauchst du nicht zu verstehen. Das Ganze wird dann zu einem **char**-Zeiger gecastet (im C-Stil, weil dies keine C++-Erfindung ist), und der Parameterliste zugewiesen. Der t-Parameter soll den Typ enthalten, von dem der optionale Parameter ist. __va_argsiz(t) hat den folgenden Aufbau:

```
#define __va_argsiz(t) (((sizeof(t) + sizeof(int) - 1) /
sizeof(int)) * sizeof(int))
```

va_end dient schließlich dazu, den Zeiger auf Null zu setzen.

Den genauen Sinn und die Funktionsweise dieser Makros musst du nicht verstehen, mach dir also keine großen Gedanken über diese Sachen.

Eine Funktion deklarieren

Damit der Compiler auch weiß, dass eine Funktion eine variable Anzahl von Parametern hat, muss die Funktion dementsprechend deklariert werden. Das geht mit drei Punkten:

```
void Funktion(int Zahl, ...);
```

In der Funktion muss dann noch die Liste deklariert werden:

```
va_list Liste;
```

Beispiel vararg1.

Du solltest es möglichst vermeiden, diesen Konstrukt in irgendwelchen Rundenspielen oder anderen Programmen anzuwenden, da der Code recht schwer zu verstehen ist. Du kannst diese Makros allerdings sinnvoll verwenden, wenn du entweder Funktionen wie printf() oder wie im Beispiel vararg1 schreibst. Es gibt eben immer wieder Spezialfälle, die diese Technik benötigen. Nochmal in aller Deutlichkeit: Soweit es dir möglich ist, solltest du es in C++ aber immer vermeiden.

Wichtig ist allerdings zu wissen, wie du selbst Makros schreiben und benutzen

kannst. Um mal einen Überblick über Unterschiede und Gemeinsamkeiten zu kriegen, schau dir folgende Tabelle an:

Vergleich	Funktion	Makro
Umfang	1 bis unendlich viele Befehle	1 bis 5 Befehle (wenn mehr, dann Funktionen verwenden)
Verlauf	Bei Aufruf springt der Compiler zu der Adresse. Bei **inline**-Funktionen keine Sprünge, weil Funktion direkt in den Code geschrieben wird.	Der Aufruf wird durch den zugehörigen Code noch vor dem Linken ersetzt. Ein Makro verhält sich dann wie eine **inline**-Funktion.
Speicher	Exe-Datei bleibt klein, bei **inline**-Funktionen wird sie entsprechend größer.	Exe-Datei wird etwas größer
Übergabe	Parameter und Rückgabe möglich	Parameter ja, Rückgabe vielleicht

Später werden sämtliche Präprozessordirektiven genauer behandelt. Dort kannst du noch ein paar Sachen für Makros lernen. Da Makros aber fehleranfällig sein können und eine alte Erfindung sind, gibt es für C++ eine bessere Möglichkeit, Funktionen für mehrere Datenytpen zu gebrauchen - besser als Funktionen überladen - die Template-Funktionen. Dennoch bleiben Makros für einige Fälle die Favoriten.

Der Zufall

Eine weitere wichtige und sinnvolle Funktion ist `rand()`. Mit dieser Funktion kannst du Zufallszahlen erhalten, was sehr wichtig bei Spielen jeglicher Art ist. Um den Zufall nutzen zu können, sind drei Funktionen in zwei unterschiedlichen Header-Dateien nötig. Zur Initialisierung ist folgendes nötig:

```
srand( static_cast<unsigned> (time(NULL)) ); //time(NULL) ist ein weiterer Funktionsaufruf
```

Diese Initialisierung (nicht im ursprünglichen Sinne) und Festlegung des Startpunktes ist notwendig, damit der Zufall korrekt und wirklich zufällig gebraucht werden kann. Um jetzt einen Zufallswert zu erhalten, muss die entsprechende Funktion ausgeführt werden:

```
int Zufall = rand();
```

Der Zufall muss nur einmal initialisiert werden und steht dann dem Gültigkeitsbereich zur Verfügung. Die zwei nötigen Header-Dateien sind **ctime** und **cstdlib**. Die maximale Größe der zurückgegebenen Zahl ist in **cstdlib** mit `0x7FFF` (also dezimal: `32767`) definiert.

Rechnerei mit rand()

Wir brauchen solche großen Werte in der Regel nicht, wollen aber den Zufall begrenzen. Das geht perfekt mit dem Modulo-Operator `%`, der den Rest einer ganzzahligen Division zurückgibt:

```
int Maximal300 = rand()%301; //hier kann die Variable einen Wert von null bis 300 annehmen
```

Um einen Mindestwert zu bestimmen, kannst du eine Zahl dazuaddieren:

```
int Minimal100Maximal300 = rand()%201 + 100; // Variable kann 100 bis 300 sein
```

Du kannst mit dieser Funktion rechnen wie du mit anderen auch rechnen kannst. Um einen zufällig gewählten Buchstaben zu erhalten könntest du folgendes mit einer **char**-Variable anstellen:

```
char GroßBuchstaben = rand()%26 + 65; // siehe ASCII-Tabelle
```

Ein Computer kann meines Wissens nach nur Pseudo-Zufallszahlen liefern. Das heißt, dass diese scheinbaren Zufallszahlen, die `rand()` liefert, nicht wirklich zufällig sind. `rand()` berechnet die vergangene Zeit einige Male mit diversen anderen Zahlen und gibt danach das Ergebnis zurück. Das Ergebnis ist also immer von der vergangenen Zeit abhängig. Echte Zufälle kann wohl nur die Natur herbeizaubern. Wenn man mehrere Male kurzzeitig hintereinander `rand()` aufruft, wirst du bemerken, wie teilweise die selben Werte zurückgegeben werden.

Zusammenfassung

Mit `printf()` hast du in diesem Kapitel erstmals eine ernsthafte Alternative zu `cout` kennengelernt. `printf()` ist nicht so eine große Funktion wie die, die mit `cout << tralala;` verbunden ist - das wirkt sich sehr positiv auf die Dateigröße aus.

Ein `getchar();` am Ende einer `main`-Funktion übernimmt genau den Part, den vorher `cin >> tralala;` eingenommen hatte. Auch hier wird die Exe nicht so mit den Klassenzeugs, das in **iostream** steht, gefüllt. Beide Funktionen bieten uns die Möglichkeit, **iostream** ganz aus dem Programm zu lassen, solange keine Werteingabe erfolgen muss.

Nach diesen Funktionen bin ich genauer auf Makros eingegangen. Zwar werden wir in einem späteren Kapitel eine sehr gute Alternative kennenlernen. Makros haben aber in einigen Spezialfällen durchaus ihre Daseinsberechtigung.

Um vielleicht auch einmal solche Funktionen wie `printf()` schreiben zu können, hab ich dir dann auch noch variable Parameteranzahlen vorgestellt. Und zwar mithilfe von Makros...

Erst der Zufall lässt Spiele so richtig zu Spielen werden. Natürlich gibt es auch hierzu eine Verbesserung zu **GDO** - mit einem Makro und vielen Zufällen. Mit den zufällig generierten Werten kann GDO ein bisschen realistischer gemacht werden.

Workshop

Fragen

1.) Wieso denn olle Kamellen aus C-Zeiten??
2.) Dieses Werk schimpft sich C++-Tutorial. Wieso wird dann nicht die **iostream**-Bibliothek bis zuletzt verfochten??
3.) `rand()` liefert nicht immer die erwartete Zufälligkeit. Gibt es Alternativen??

Programme

1.) Schreibe eine Funktion, die beliebig viele Parameter als **char**, **int** oder **float** ausgibt! Welchen Datentyp die Funktion benutzen soll, muss der Benutzer entscheiden!

Kapitel 15

In diesem Kapitel geht weiter mit nützlichen Dingen. Zu allererst steht die Dateibehandlung mit C-Funktionen auf dem Plan. Dann werden wir mit den Bitoperatoren Variablen anders nutzen lernen. Am Ende zeige ich kurz das Schlüsselwort **auto** und nach diesem letzten Schlüsselwort aus C kannst du dann einige weitere C-Funktionen in der Referenz der Bibliotheksfunktionen kennen lernen.

Dateien

Man unterscheidet größtenteils zwischen Text- und Binärdateien. Textdateien haben zu Binärdateien einen feinen Unterschied, den du in diesem Kapitel mitbekommen wirst.

Textdateien

Diese Dateien enthalten hauptsächlich Buchstaben, Zahlen und Zeichen, die du als Wörter oder zusammenhängenden Text lesen kannst. Sämtliche Zeichen sind aus dem ASCII-Code entnommen. Um die folgenden Funktionen nutzen zu können (alle sind in **cstdio** definiert), muss es einen Zeiger auf eine Struktur geben, die von den Funktionen genutzt wird. Ein Zeiger vom Typ `FILE`:

```
FILE *Datei;
```

Zum Öffnen einer Datei kannst du die Funktion `fopen()` benutzen, die folgenden Aufbau hat:

```
FILE* fopen(const char* FileName, const char* Mode);
```

Als erstes Argument kannst du einen einfachen Dateinamen angeben, dann wird im Verzeichnis des Programms nach der Datei gesucht. Oder du gibst einen kompletten Pfad an.

Mit dem zweitem Argument kannst du die Art der Verwendung durch eine Zeichenkette oder einen Buchstabe bestimmen:

Zeichenkette	Verwendung der Datei	Muss die Datei vorhanden sein?
r	Lesen	Ja
r+	Lesen und Schreiben	Ja
a	Hinzufügen	Wenn nicht, wird eine Datei erzeugt
a+	Hinzufügen und Lesen	Wenn nicht, wird eine Datei erzeugt
w	Schreiben bzw. Überschreiben	Wenn nicht, wird eine Datei erzeugt
w+	Schreiben bzw. Überschreiben	Wenn nicht, wird eine Datei erzeugt

`fopen()` gibt einen Zeiger auf eine Struktur des Typs `FILE` zurück, die in der Funktion erstellt wurde. Falls jedoch ein Fehler aufgetreten ist, gibt die Funktion `NULL` zurück. Nachdem die benötigte Struktur erstellt wurde und ein Zeiger auf diese zurückgegeben wurde, kann aus dieser gelesen

werden, und zwar mit `fgets()`. Diese Funktion liest aus einer Datei Zeichen, bis das Steuerzeichen `\n` kommt. Sie hat den Aufbau:

```
char* fgets (char* Buffer, int BufferSize, FILE* file);
```

`char* Buffer` ist ein Zeiger auf den String, in den die gelesenen Zeichen gespeichert werden sollen. `BufferSize` gibt die Anzahl der zu lesenden Zeichen an. `FILE *file` ist der Zeiger auf die Struktur, die mit `fopen()` erstellt wurde. Um die Datei wieder zu schließen, verwendet man `fclose()`, deren einziges Argument der Zeiger auf die `FILE`-Struktur ist.

Um auch aus Dateien lesen zu können, deren Größe dir unbekannt ist, kannst du die Funktion `feof()` verwenden. Sie verlangt den Zeiger auf die `FILE`-Struktur und gibt `NULL` zurück, wenn das Dateiende erreicht ist.

Beispiel read1.

In der `FILE`-Struktur wird die Position, an der zuletzt gelesen oder geschrieben wurde, gespeichert. Damit wir darauf Zugriff haben können, gibt es die Funktion `fgetpos()`:

```
int fgetpos (FILE* GetPosition, fpos_t* pos);
```

`GetPosition` ist der Zeiger auf die `FILE`-Struktur und `pos` ist ein Zeiger auf eine Variable vom Typ `fpos_t`, der als neuer Typ von **`long`** definiert wurde. `pos` gibt die Position des letzten Zugriffs in der Datei an. Um die Position selbst festzulegen, kannst du `fsetpos()` aufrufen, die den selben Aufbau wie `fgetpos()` hat. `pos` gibt dann an, an welche Stelle der Datei zunächst zugegriffen werden soll.

Beispiel read2.

In die Datei TO2.txt kannst du die zu öffnenden Dateien reinschreiben. Jeder Dateiname muss in einer eigenen Zeile stehen.

Damit du auch etwas in eine Datei schreiben kannst, musst du `fprintf()` benutzen:

```
int fprintf(FILE* WriteTo, const char* Format, ...);
```

`WriteTo` ist der Zeiger auf die `FILE`-Struktur aus `fopen()`, `Format` gibt die Formatierung bekannt, die sich nach gleichen Regeln wie bei `printf()` verhält. Durch das Konzept mit der variablen Argumentzahl können beliebig viele Variablen übergeben und dann in die Datei geschrieben werden.

Beispiel write1.

Neben den obigen Funktionen gibt es noch weitere in der Headerdatei stdio.h. Um einen einzelnen Buchstaben zu schreiben kannst du auch `fputc(char Buchstabe, FILE* WriteTo)` verwenden. Um ein String ohne Variablen in die Datei zu schreiben, kannst du `fputs(char *String, FILE *WriteTo)` nehmen, und für das Auslesen einzelner Buchstaben `fgetc(FILE *ReadFrom)`, wobei du beachten musst, dass der

zurückgegebene **int**-Wert erst zum gewünschten Zeichen gecastet werden muss (etwa `cout << static_cast<char>(fgetc(Datei));`).

Wie Strings müssen auch Dateien mit dem `NULL`-Zeichen versehen werden. Manche Funktionen schreiben das `NULL`-Zeichen in die Datei, manche nicht. Du kannst das so hinkriegen:

```
//...
//Datei-Kursor steht schon am Ende der Datei
if(fgetc(file1) != '\0') fputc('\0', file1);
```

Weiteres zu fopen()

Zu allen Modi von fopen gibt es noch eine Binär-Version:

Textmodus	Binärmodus
r	rb
r+	r+b oder rb+
a	ab
a+	a+b oder ab+
w	wb
w+	w+b oder wb+

Wenn aus einer im Textmodus geöffneten Datei gelesen wird, dann werden (von allen Datei-input/output-Funktionen) CarriageReturn-Linefeed-Sequenzen (ist eine Folge aus den Zeichen 13 und 10 des ASCII-Codes bzw. `\r\n`) in einfache Zeilenumbrüche (`\n`)umgewandelt. Wenn in eine im Textmodus geöffneten Datei geschrieben wird, werden einfache Zeilenumbrüche in CarriageReturn-Linefeed-Sequenzen. Im Binärmodus wird diese Umformung nicht vollführt. So würde die Zeichenkette

```
"Hallo!!\n\nWie geht es dir??\r\nMir geht\'s gut!!"
```

im Textmodus gespeichert so aussehen:

```
"Hallo!!\r\n\r\nWie geht es dir??\r\r\nMir geht\'s gut!!"
```

während im Binärmodus das Ganze so aussehen würde, wie das Original:

```
"Hallo!!\n\nWie geht es dir??\r\nMir geht\'s gut!!"
```

Das ist ein Unterschied von 3 Zeichen.

Bei einem Linefeed (`\n`) wird der Kursor auf dem Bildschirm am Anfang der nächsten Zeile positioniert. Bei einer CarriageReturn-Linefeed-Sequenz wird der Kursor auf dem Bildschirm zunächst an den Anfang der aktuellen Zeile gerückt, anschließend eine Zeile nach unten verschoben. Die Bedeutung von `\n` und `\r\n` ist also eigentlich die selbe.

Du musst Binärdateien (die ja bekanntlich auf *.bin enden) nicht unbedingt im Binärmodus öffnen!

Lesen und Schreiben in Binärdateien

Binärdateien haben die Endung *.bin. Während in Textdateien Texte gespeichert wurden, werden in Binärdateien Datenblöcke gespeichert. Du kannst natürlich trotzdem Textdateien mit der Endung *.bin versehen. Wenn auf Datenböcke zugegriffen werden, kann das allerdings nicht mit den Funktionen `fprintf()` und `fgets()` passieren, hierfür gibt es die Funktionen `fread()` und `fwrite()`:

```
size_t fwrite(const void* Buffer, size_t SizeOfBuffer, size_t
Count, FILE* WriteTo);

size_t fread(void* Buffer, size_t SizeOfBuffer, size_t Count, FILE*
ReadFrom);
```

`size_t` ist ein als vorzeichenloser **`int`** definierter Typ, der dazu verwendet wird, um Größeninformationen zu beinhalten. Beide Funktionen geben zurück, wie viel sie geschrieben/gelesen haben. `Buffer` ist ein Zeiger zu der Variable, zu der geschrieben/von der gelesen werden soll (bei `fwrite()` ist er konstant, weil sein Wert nicht geändert wird). `SizeOfBuffer` gibt an, wie groß die Variable `Buffer` ist. `Count` gibt die Anzahl an, wie viele Male der Schreib- oder Lesevorgang erfolgen soll. Der `FILE`-Zeiger ist schließlich wieder der Zeiger zur Struktur von `fopen()`.

Beispiel bin1.

Falls du mal die Datei bin1.bin (mit dem Texteditor) öffnest, wirst du ziemlich sinnlose Zeichen erkennen. Über die Datei ist zu sagen, dass sie im Vergleich zu einer Textdatei viel größer ist, was sich besonders bei Binärdateien mit vielen Datenblöcken bemerkbar macht.

Zu diesem Kapitel soll es keine weitere Version von GDO geben, schließlich gab es bis jetzt schon vier Versionen. Die oben vorgestellten Funktionen kannst du dazu nutzen, um etwa nach Kämpfen dem Spieler die Möglichkeit zu geben, seinen Fortschritt zu speichern, oder die Gegnerdaten in externe Dateien zu bringen. Damit sollte es möglich sein, das Spiel noch dynamischer zu machen.

Du kannst zur besseren Datensicherheit eine Funktion entwickeln, die die Daten verschlüsselt, etwa zu jeder Zahl oder zu jedem Zeichen 5 dazuaddieren (so wird aus '`H`' ein '`M`') - übrigens Caesar-Chiffre genannt. Damit kannst du verhindern, dass ein Spieler, der an dem Projekt nichts entwickelte, irgendwie beschummelt. Das Hauptprogramm müsste dann mit dem passenden Entschlüsselungsverfahren die Daten korrekt entschlüsseln. Mehr zum Ver- und Entschlüsseln wirst du in einem späteren Kapitel erfahren.

Flags

Mit sogenannten Flags können gleich mehrere Informationen in einem Byte zum Argument einer Funktion werden. So können 8 verschiedene Wahr-/Falschwerte durch einen **`char`**-Parameter ersetzt werden. Hier sind Kenntnisse über die Bitoperatoren nötig, die du sicherlich schon wieder vergessen hast. Jedes Bit steht für eine Zahl die 2 hoch Index (... 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0). Wenn also die Bits geschickt verknüpft werden, diese Technik benutzen:

```
#define Fl_0 1
#define Fl_1 2
#define Fl_2 4
#define Fl_3 8
```

```
#define Fl_4 16
#define Fl_5 32
#define Fl_6 64
#define Fl_7 128

void FlagFunc(char Flag1)
{
   //Wenn Flag1 den ersten Bit auf 1 hat, so wird Flag_0 der Wert 1
zugewiesen
   int Flag_0 = Flag1 & Fl_0;

   //Wenn Flag1 den zweiten Bit auf 1 hat, so wird Flag_1 der Wert
2 zugewiesen
   int Flag_1 = Flag1 & Fl_1;
   int Flag_2 = Flag1 & Fl_2;
   int Flag_3 = Flag1 & Fl_3;
   int Flag_4 = Flag1 & Fl_4;
   int Flag_5 = Flag1 & Fl_5;
   int Flag_6 = Flag1 & Fl_6;

   //Wenn Flag1 den achten Bit auf 1 hat, so wird Flag_7 der Wert
128 zugewiesen
   int Flag_7 = Flag1 & Fl_7;

   cout << "Alle Flags ergeben zusammen: "
        << Flag_0 + Flag_1 + Flag_2 + Flag_3 + Flag_4 + Flag_5
+ Flag_6 + Flag_7 << " :)\n";
}
```

Zuerst werden die Flags definiert. Wenn nur ein Bit wahr sein soll, muss der Flag genau den Wert haben, der 2 hoch Index ist. Soll es ein Flag geben, das ein oder mehrere Flags voraussetzt, oder das eine Zusammenfassung mehrerer Flags ist, so müssen die jeweiligen Werte addiert werden:

```
#define Fl_4_5_6_7 Fl_4 + Fl_5 + Fl_6 + Fl_7
#define Fl_2_3 Fl_2 + Fl_3
```

Der Aufruf einer solchen Flag-Funktion sieht dann so aus:

```
int main(void)
{
   FlagFunc(95);
   FlagFunc(Fl_1 | Fl_2 | Fl_5 | Fl_4 | Fl_0);
   FlagFunc(Fl_4_5_6_7 | Fl_4_5_6_7 | Fl_4_5_6_7);
   //...

   return 0;
}
```

Als erstes wird das Bitmuster `01011111`, also `95` übergeben. Als nächstes wird `00110111`, also `55` und dann `11110000`, also `240` übergeben.

Beispiel flags1.

Das Prinzip der Flags wirst du oft in Windows- und DirectX-Programmierung antreffen. Hierfür reicht es allerdings zu wissen, dass mehrere Flags mit dem ODER-Operator `|` verknüpft werden müssen.

Das Schlüsselwort auto

Mit diesem Abschnitt ist auch das letzte Schlüsselwort der Teilmenge C in C++ geklärt. Nur der Vollständigkeit halber.

Das **auto**-Schlüsselwort gibt an, dass eine Variable lokal gespeichert ist. Sie ist nur in Funktionen verfügbar.

```
void Function(void)
{
   auto int A = 500;
   int A2 = 5444;
}
```

Es besteht kein Unterschied zwischen den zwei Variablen, da sie den selben Gültigkeitsbereich haben. Du brauchst **auto** nicht verwenden, es ist eigentlich nicht wirklich sinnvoll. Falls du jemals dieses Schlüsselwort angewandt siehst, weißt du bescheid.

Die C-Standardbibliothek

Hier kannst du dir Erläuterungen und Beispiele zu einigen vielen weiteren Funktionen der Standard-Bibliothek anschauen:

Runtime Library Reference

Der Abschnitt soll dir zum Stöbern dienen, damit du (zu `rand()` und `printf()`) weitere Standardfunktionen kennenlernst. Leg besonders auf die Funktionen in **ctime**, **cstring** und **cstdlib** ein Auge.

Zusammenfassung

In diesem und im letzten Kapitel hast du viel über C kennengelernt. Dadurch sollst du erst einmal mit einfachen Dingen an die Datenausgabe auf Bildschirm und in Dateien herangeführt werden. Die objektorientierten Techniken in C++ sind um einiges schwieriger zu erlernen, sodass ich die **iostream**-Bibliothek erst einmal auf ein späteres Kapitel schiebe.

Mit den Dateibehandlungsroutinen kannst du jetzt Speicherstände, Datenbankeinträge, Logbücher und so weiter dauerhaft auf die Festplatte bannen. Die vielen Funktionen, die dafür nötig sind, erscheinen natürlich zunächst etwas verwirrend. Mit entsprechender Übung hat sich das aber bald eingewöhnt.

Vergiss den kleinen Unterschied zwischen Text- und Binärdateien nicht. Auch hier ist es zunächst nicht leicht, Hand und Fuß zu fassen. Du könntest dir doch einmal die Aufgabe stellen, GDO mit Laden und Speichern auszurüsten!

Flags zielen nun wieder in eine ganz andere Richtung. Anstatt 8 **bool**-Variablen zu machen, reicht es doch eigentlich, eine **char**-Variable mit den Bitoperatoren so zu manipulieren, dass das Ergebniss dasselbe ist. Wie gesagt, die WinAPI (Windows Applikation Programming Interface) und DirectX machen von dieser Technik Gebrauch in Massen.

Mit **auto** ist das letzte C-Schlüsselwort geklärt, es folgen nun nur noch C++-spezifische Schlüsselwörter. **auto** ist eigentlich jede Variable, die in einer Funktion erschaffen wird. Damit ist, so wie's aussieht, **auto** überflüssig.

Obwohl der Abschnitt "Die C-Standardbibliothek" in diesem Kapitel nur kurz ist, kommt ihm große Wichtigkeit zu. Dem Link gefolgt, siehst du einige weitere Links zu Funktionsbeschreibungen der Standardheader. Du solltest dich mal kräftig mit den Funktionen darin auseinandersetzen, denn es lohnt sich wirklich!

Workshop

Fragen

1.) Was könnte der Grund sein, die in der Zusammenfassung genannten 8 **bool**s durch 1 **char** zu ersetzen und mithilfe von Bitoperatoren zu manipulieren??
2.) Wieso erfindet man ein Schlüsselwort, das eigentlich total sinnlos ist?? Und wieso kommt es in diesem Tutorial??
3.) Gibt es in der Funktionsbibliothek von C++ auch Funktionen, um die Vorteile der Dateisysteme nutzen zu können??

Programme

1.) Schreibe ein Programm, das mit Kommandozeilenparametern eine Datei kopiert. Führe das Programm von der Eingabeaufforderung aus und übergebe die Parameter. Im Programm selber soll der erste Parameter die Quelle sein und der zweite Parameter das Ziel. Varianten:

a.) mit den in diesem Kapitel vorgestellten Funktionen
b.) mit einer Funktion aus **cstdio**, die das Kopieren voll und ganz übernimmt.

2.) Dasselbe ohne Kommandozeile. Der Benutzer soll ins Programm eingeben können, welche Datei er kopiert haben will.

Links für Dateibehandlung von C++:

http://www.volkard.de/vcppkold/dateien.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop14_7.html
http://www.schornboeck.net/ckurs/dateien.htm

Kapitel 16

Ab diesem Kapitel folgen einige Kapitel, die sich ausschließlich mit den Sprachelementen beschäftigen, um die C erweitert wurde. C++ bietet viele gute und bereichernde Möglichkeiten, um das Programmieren zu erleichtern. Du wirst sicher schon überall gehört haben, dass C++ gegenüber C das objektorientierte Programmieren berherrscht. Tatsächlich bringt dieser Ansatz viele Vorteile.

Objektorientierte Programmierung

Objektorientierung ist eigentlich mehr eine Philosophie als ein Programmieransatz. Wenn du mal um dich rumschaust, lässt sich eigentlich alles in Objekte gliedern. Der Stuhl, auf dem du hockst, den Bildschirm, den du anstarrst, das Glas Senfgurken, das im Kühlschrank steht. Jedes dieser Objekte und generell jedes Objekt hat bestimmte Eigenschaften und Möglichkeiten.

Im Supermarkt steht ein Regal voller Konservendosen. Fast alle Dosen haben die Gemeinsamkeit, dass sie aus dem gleichen Material bestehen. Es gibt große, kleine, und mittelgroße Dosen, unterschiedlich breit und hoch. Aber auch wieder jede Dose hat einen unterschiedlichen Inhalt. Es entsteht ein Muster, nach dem wir uns eine Dose vorstellen. Wenn dich jemand dumm fragt, wie denn eine Dose aussieht, antwortest du prompt: "Ein Zylinder aus Blech, so etwa 6 bis 10 cm hoch und 4 bis 8 cm breit. Auf der runden Seitenfläche ist ein Aufkleber aufgeklebt. In der Dose drin ist das, was auf dem Aufkleber beschrieben ist. Den Rest kannst du dir ja wohl denken, du Nase!!".

So haben wir die Dinge in diesem Tutorial bisher nicht betrachtet. Wie eben das Muster, lassen sich Objekte verallgemeinern. Und zwar in Klassen. In einer Klasse werden zusammengehörende Eigenschaften und Funktionsmöglichkeiten zusammengefasst. Vor ein paar Kapiteln hast du komplexe Datentypen kennengelernt, genau dazu gehört auch die Klasse. Du kannst sie dir als stark erweiterte Struktur vorstellen.

Struktur vs. Klasse

Strukturen sind lediglich Container für eine Vielzahl von Variablen. Mit den Daten sind Funktionen nicht direkt verbunden. Jeder kann alles mit den Daten machen. Kontrolle darüber, was mit diesen Daten passiert, ist dir nur schwer möglich. Durchaus vergleichbar ist das mit einem Stapel von vertraulichen Akten, die kreuz und quer in einer Stadt verteilt werden (na gut, so heftig ist das nun auch wieder nicht).

Klassen sollen jetzt nicht als das Paradies oder den Programmiererhimmel dargestellt werden. Auch mit Klassen und Objekten kann man sich's ziemlich herb zerbomben. Aber es deuten sich schon einige Verbesserungen diesbezüglich gegenüber Strukturen an.

Klassen enthalten sowohl Daten als auch die zur Bearbeitug vorgesehenen Funktionen. Wichtige Daten, eigentlich alle Daten - so sieht es der OOP'ler vor - dürfen von außerhalb der Klasse nicht erreichbar sein. Denn dazu dienen die öffentlich verfügbaren Funktionen. Du als Programmierer kannst also genau kontrollieren, was mit deinen Daten passiert.

Objekte

Wie eine Variable eine Instanz eines Variablentyps ist, so ist ein Objekt eine Instanz einer Klasse. Fast alle Dosen haben ein Blechgehäuse. Fast jede Dose hat aber einen anderen Inhalt und andere Maße. Das Objekt `ErascoSuppentopf` ist anders als das Objekt

AprikosenGestueckelt. Jedes Objekt hat seine Eigenart, aber immer in dem Rahmen, den die Klasse - die Allgemeinheit vorgibt.

Eine Dose hat sicherlich nicht allzu viele Fähigkeiten. Vielleicht Auslaufen() oder Fuellen(). Ich nehme noch ein weiteres Beispiel - ein Haus. Einige sinnvolle Variablen, in OOP **Attribute** oder **Member** genannt, sind die Anzahl der Türen, Anzahl der Fenster, Anzahl der Räume und Wohnraum. Sinnvolle **Methoden** oder **Memberfunktionen** (sind Funktionen in Klassen) wären dabei Türen und Fenster öffnen(), den Wohnraum erweitern(), das Haus einreißen() und so weiter.

Eine Klasse definieren

Für die Definition einer Klasse ist das Schlüsselwort **class** zuständig. Die Definition einer primitiven Klasse ist der Definition einer Struktur ziemlich ähnlich:

```
class House
{
   short m_Doors;
   short m_Windows;
   short m_Rooms;
   short m_Space;
};
```

Diese Klasse enthält nur 4 Attribute. Diese können nicht in der Klassendefinition initialisiert werden (was bei einer Struktur ebenfalls nicht möglich ist).

Regeln zur Benennung

Auch wenn dieser Code ungewöhnlich aussieht, solltest du dich an folgende Namensregeln gewöhnen:

- Englische Namen!!!
- Attribute besonders hervorheben; das "m_" (für member) hat sich eingebürgert.
- für Zeiger ein "p" (fpr pointer)voranstellen.
- kein "_" als erstes Zeichen nehmen.

Ansonsten gelten die Namensregeln wie bei den Variablen aus Kapitel 2 oder 3!

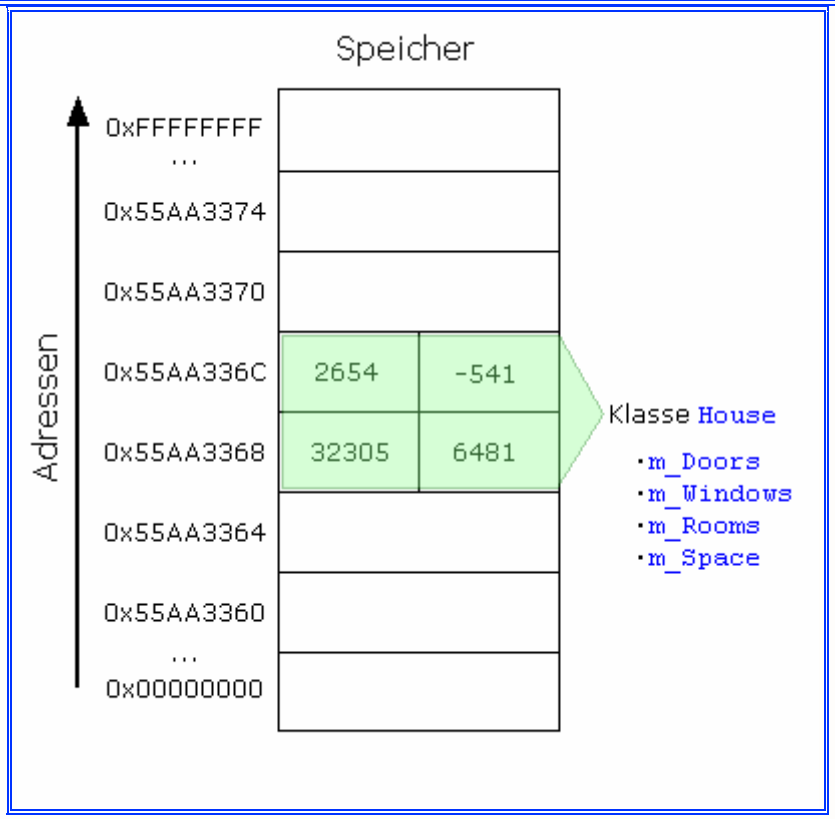
Ein Objekt daraus deklarieren

Bei der Definition einer Klasse wird die Klasse sowie deren Attribute und Methoden dem Compiler nur bekannt gemacht, aber noch nicht mit Speicher verbunden, deswegen kann hier auch kein Anfangswert gegeben werden. Um Speicher für ein Objekt zu reservieren, kannst du die gewöhnliche Variablendeklaration verwenden:

```
House WeißesHaus;
House GruenesHaus;
House Vogelkaefig;
House *pHouse = &WeißesHaus;
```

House im Speicher

Bis jetzt würde ein Objekt der Klasse House im Speicher nicht anders aussehen als ein gleichartiges Gebilde mit der Struktur als Grundlage:



Die vier Werte standen natürlich wieder gerade im Speicher. Insgesamt ist die `House`-Klasse schon 8 Byte groß.

Zugriff auf Klassenelemente

Wenn du in einem Programm versuchst, auf eine der Variablen mit dem Zugriffsoperator Punkt '`.`' zuzugreifen, wirst du eine Fehlermeldung erhalten. In dieser wird es wahrscheinlich um den Zugriffsschutz gehen: "short WeißesHaus::m_Space is private". Dies ist die typische Fehlermeldung für Zugriff auf ein nicht öffentliches Element.

Dieses "**`private`**" ist ein Schlüsselwort von C++ für den Zugriffsschutz. Es bedeutet, dass eine Methode oder ein Attribut nicht willkürlich vom Programmierer, aber von Objekteigenen Methoden aufgerufen bzw. verändert werden kann. Attribute und Methoden einer Klasse werden automatisch **`private`**, wenn nichts anderes angegeben ist. Deswegen kann auf `WeißesHaus.m_Space` noch nicht zugegriffen werden.

Im Sinne der OOP sind eigentlich ALLE Attribute einer Klasse **`private`**. Für die bereits bestehenden wollen wir später Funktionen definieren, die mit diesen Attributen arbeiten können. Außerdem soll noch ein weiteres Attribut hinzukommen: **`short`** `m_Persons` - auch wieder **`private`**.

In der Definition von oben waren also alle Attribute implizit **`private`**. Hier jetzt die Definition, wo wir die Attribute implizit und explizit **`private`**-isieren:

```
class House
{
   short m_Doors;       //automatisch unausdrücklich(implizit)
private
   short m_Windows;
   private:
      short m_Rooms;    //ausdrücklich(explizit) private
```

```
      short m_Space;
      short m_Persons;
};
```

Methoden/Memberfunktionen

Methoden gehören meistens in den öffentlich zugreifbaren Bereich. Den öffentlichen Bereich markiert das Schlüsselwort **public**.

> Da die meisten Attribute **private** sind, aber trotzdem nach außen hin zugänglich sein sollen, muss man Methoden schreiben, um auf diese Elemente zugreifen zu können. Weit verbreitet ist das hübsche Päärchen `Set...()` und `Get...()`. Einfach "Set" oder "Get" vor den Namens des Attributes und fertig ist der Name für die Funktion. Diesen wenden wir uns weiter unten zu.

In die Klassendefinition kommt der Methodenprototyp. Die Implementation der Funktion kann entweder in die Klasse selbst rein (gleich am Prototypen dran) oder extern definiert werden. Folgendes Beispiel implementiert eine Funktion zum Erweitern des Hauses und eine andere Funktion zur statistischen Darstellung des Hauses:

```
class House
{
   //Memberdeklarationen ...
   public:
      void Extend(short Rooms, short Space) //interne Definition
      {
         m_Rooms += Rooms;
         m_Space += Space;
      }
      void Statistics(void);            //Prototyp deklariert,
Definition ist aber extern
};

void House::Statistics(void)
{
   cout << "Das Haus hat "<< m_Doors << " Tueren\n"
        << m_Windows << " Fenster\n"
        << m_Rooms << " Raeume\n"
        << m_Space << " Kubikmeter Wohnraum\nund es sind gerade "
        << m_Persons << "Leute drin.";
}
```

Zuerst wird Funktion `Extend()` deklariert und auch gleich definiert. Dies nennt man interne Definition. Interne Definitionen sollest du nur bei Methoden anwenden, die sich problemlos in einer Zeile implementieren lassen.

Für `Statistics()` wird zunächst nur der Prototyp deklariert. Der Funktionsrumpf wird zunächst nach außen verlagert. Das hilft, den Code der Klassendefinition übersichtlich zu halten.

> **So wird's gemacht!**
>
> Merk dir: Eine Klassendefinition sollte immer nur die Deklarationen der Attribute und Methoden enthalten. Eine Klassendeklaration ist nur im Sonderfall nötig, wenn zwei Klassen in ihren Definitionen sich gegenseitig benötigen. Ein Projekt mit einer Klasse besteht wegen guten Programmierstils aus 3 Sourcedateien:
>
> 1.) Einen Header zur Definition der Klasse.
> 2.) Eine cpp-Datei zum Definieren der Methoden dieser Klasse.

3.) Eine cpp-Datei für den Rest: `main`-Funktion und Objektinstanzierung.

Jede Klasse soll also eine Datei für die Klassendefinition (etwa **Dose.h** oder **Haus.h**) und eine Datei, in denen die Methoden vollständig programmiert werden (**Dose.cpp** bzw. **Haus.cpp**), enthalten.

Methoden extern definieren

Allem voran steht der Rückgabetyp. Damit der Compiler die Methode der richtigen Klasse zuordnen kann, kommt vor den Funktionsnamen der Bereichsoperator `::` und der Name der Klasse. Schnell noch die Parameterliste hinten angeklebt, dann kommt endlich der Funktionsrumpf. Noch einmal der Code zu `Statistics()`:

```
void House::Statistics(void)
{
   cout << "Das Haus hat "<< AnzahlTueren << " Tueren\n"
        << AnzahlFenster << " Fenster\n"
        << AnzahlRaeume << " Raeume\n"
        << Wohnraum << " Kubikmeter Wohnraum\nund es sind gerade "
        << AnzahlPersonen << "Leute drin.";
}
```

Innerhalb der Methoden der Klassen kannst du, egal welche Sicherheitsstufe, auf die Attribute und Methoden zugreifen.

Nachrichten senden

Streng genommen heißt Methoden aufzurufen eine Nachricht an das Objekt zu senden. Wie das nun geht?? Eigentlich so, wie du bei Strukturen auf deren Elemente zugreifst. Zugriff auf (Attribute (falls es öffentliche Attribute gibt) sowie) Methoden erhältst du Klassenextern mit dem Zugriffsoperator Punkt '`.`' und bei Zeigern mit `->`. Bei den Methoden kommt noch die Parameterliste dran:

```
House WeißesHaus;
House GruenesHaus;
House Vogelkaefig;
House *pHaus = &WeißesHaus;

pHaus->Statistics();
GruenesHaus.Extend(2, 300);
```

Set-und Get-Methoden

Viele der Attribute einer Klasse sind zu recht **`private`**. Um dennoch auf diese Elemente zugreifen zu können, hat man sich auf eine einheitliche Namensgebung für die zuständigen Funktionen geeinigt. Um die Werte der Attribute zu ändern, setzt man "Set" als Präfix des Funktionsnamens. Um Attribute nur zu lesen, kommt "Get" in frage. Hier für die Klasse `House` ein paar Methoden:

```
class House
{
   private:
      short m_Doors;
      short m_Windows;
      short m_Rooms;
      short m_Space;
      short m_Persons;
   public:
      void Extend(short Rooms, short Space);
```

```
      void Statistics(void);

      short GetDoors(void);
      void SetDoors(short Doors);

      short GetWindows(void);
      void SetWindows(short Windows);

      short GetRooms(void);
      void SetRooms(short Rooms);

      short GetSpace(void);
      void SetSpace(short Space);

      short GetPersons(void);
      void SetPersons(short Persons);
};

//Definitionen von Extend() und Statistics()

short House::GetDoors(void)
{
   return m_Doors;
}

void House::SetDoors(short Doors)
{
   if(Doors >= 0) m_Doors = Doors;
}

//Definitionen der restlichen Set- und Get-Funktionen
```

Klingelt's?? Bei einer normalen Zuweisung würde `m_Doors` ohne irgendeine Kontrolle einen Wert erhalten:

```
//ungeschützt:

Vogelkaefig.m_Doors = rand()-16000;

//geschützt:

Vogelkaefig.SetDoors(rand()-16000);
```

Die ungeschützte Variante lässt negative Zahlen zu. Da ein Haus mit weniger als Null Türen aber nur schwer vorstellbar ist, sollte man das einfach nicht zulassen. Durch den Zugriffsschutz sind wir also gezwungen eine Methode dafür zu schreiben. Wieso nicht gleich mit einem sinnvollen Wertebereich?? `SetDoors()` ist so ausgelegt und lässt nur positive Werte durch. Schätzungsweise fällt die Hälfte der Zuweisungen mit `SetDoors()` weg.

Moment mal!! Ist der Wohnraum nicht abhängig von den im Haus vorhandenen Räumen?? Keine Räume - kein Wohnraum:

```
void House::SetRooms(short Rooms)
{
   if(Rooms > 0) m_Rooms = Rooms;
   else if(Rooms == 0)
   {
      m_Rooms = 0;
      m_Space = 0;
```

```
    }
}

void House::SetSpace(short Space)
{
   if(Space > 0) m_Space = Space;
   else if(Space == 0)
   {
      m_Space = 0;
      m_Rooms = 0;
   }
}
```

In solchen Änderungsfunktionen darfst du nicht `SetSpace()` bzw. `SetRooms()` nehmen, sonst würde es zu endloser Rekursion kommen:

```
void House::SetRooms(short Rooms)
{
   if(Rooms > 0) m_Rooms = Rooms;
   else if(Rooms == 0)
   {
      SetRooms(0); //falsch
      SetSpace(0); //falscher
   }
}

void House::SetSpace(short Space)
{
   if(Space > 0) m_Space = Space;
   else if(Space == 0)
   {
      SetSpace(0); //am falschesten
      SetRooms(0); //oberfalsch
   }
}
```

Dann ist allerdings auch kein Fenster, keine Tür und keine Person im Haus.

Beispiel house1.

Zusammenfassung

Ein entscheidender Schritt in Richtung OOP ist getan. Du weißt nun bescheid, was Klassen und Objekte sind, was Klassen so alles können und wie der Zugriffsschutz einzusetzen ist. Und dennoch sind dies nur grundlegende Schritte, denn C++ hat noch einiges auf Lager.

Eine Klasse ist die Verallgemeinerung vieler Objekte mit ähnlichen oder gleichen Eigenschaften. Wie bereits ausgehandelt, könnte es eine Klasse `Dose` und eine Klasse `House` geben. Ein Objekt ist eine Instanz einer Klasse - für `House` könnte das ein `Schloss` sein.

Damit nicht jeder auf die Daten zugreifen kann, gibt es die Schlüsselwörter **`private`** und **`public`**. **`private`** lässt Elemente der Klasse in den Atomschutzbunker einliefern, wo nur die Methoden der Klasse rein können. **`public`** dagegen befördert Elemente einer Klasse an die Erdoberfäche. Im Zusammenhang mit der Vererbung wirst du noch ein weiteres Schlüsselwort, das den Zugriffsschutz regelt, zu nutzen lernen.

Wie das nunmal so ist, hat man für die schon bekannten Dinge neue Namen erfunden. Damit du nicht vorzeitig an diesen Klippen zerschellst, hier die "Übersetzungen":

normal-C++	OOP
Variable, Platzhalter, Schublade (so werden die in manchen Tutorials genannt)	Member, Membervariable, Attribut
Funktion	Memberfunktion, Methode, Dienst
Instanz eines komplexen Datentypes (etwa **struct**, **union**)	Objekt als Instanz einer Klasse
Funktionsaufruf	Nachricht senden, Methode aufrufen

Wichtig ist, die Deklarationen und Definitionen der Methoden zu trennen. Bei umfangreichen Klassen ist einfach kein Platz für eine Methodendefinition! Also nochmal: Klassendeklaration in **KlasseXYZ.h** und Methodendefinitionen für die Klasse in **KlasseXYZ.cpp**!

Auch hier gilt es, die Materie zu verstehen und mit beiden Füssen auf dem neuen Land zu stehen! Probiere mit den Klassen rum; versuch, Programme, die du vorher programmiert hast, mit Klassen zu verbessern oder ganz auf Klassen und Objekte zu stützen.

Workshop

Fragen

1.) Gibt es das nicht auch in C??
2.) Was ist besser anzuwenden, wenn es nur um Daten geht, nicht um Funktionen - Struktur oder Klasse??
3.) Wieso muss man für jedes Attribut extra Funktionen erfinden, nur um den Wert zu bekommen oder zu ändern??
4.) Ist **private** wirklich so sicher, dass niemand von außen an private Elemente rankommt??
5.) Das Objekt `cout` - wieso gab es noch keinen richtigen Funktionsaufruf, sondern nur das Gefummle mit dem `<<` ??
6.) Kann man das **private** weglassen, wenn Elemente einer Klasse doch sowieso **private** sind??
7.) Kann man mehrere Bereiche in der Klasse **public** und **private** machen??
8.) Was ist hier falsch??

```
class Car
{
   int Speed;
   int MaxSpeed;
}
```

Programme

1.) Implementiere eine Klasse `Book`, die Daten über Bücher speichert (ISBN, Preis, Seitenzahl, Author(en), Bindungs- und Umschlag-Art, Titel, Vorrat im Lager, Verlag). In einem Test-Programm soll der Benutzer beliebig viele Bücher erstellen können und die Daten eingeben. Fange mit einem Attribut an und füge nach und nach immer mehr Attribute ein.
2.) Erweitere das Programm so, dass der Benutzer auf Wunsch seine kleine Datenbank in einer Datei speichern kann.
3.) Versuche dasselbe mit der Klasse `House`. Der Benutzer soll die Verwaltungsgebäude seiner Stadt in die Datenbank aufnehmen. Erweitere die `House`-Klasse entsprechend!
4.) Lege eine Klasse an, die die Funktionalität der C-Dateibehandlungs-Routinen übernimmt. Jede C-Funktion soll in einer Methode aufgerufen werden, die die Parameter übernimmt und weitergibt.

Links:

http://www.volkard.de/vcppkold/klassen.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop9.html
http://www.schornboeck.net/ckurs/klassen.htm

Kapitel 17

Hier werde ich dir einige weitere Möglichkeiten zeigen, was du mit Klassen alles anstellen kannst. Eine wichtige Errungenschaft ist der Konstruktor. In vielen Fällen ebenfalls sehr wichtig ist der Destruktor. Außerdem lernst du hier die Schlüsselwörter **this**, **mutable** und **explicit** kennen sowie die weiteren Bedeutungen von **const** und **static**. **inline** bildet in diesem Kapitel den Abschluß.

Der Konstruktor

Ein Konstruktor ist die Methode, die bei jedem Objekt zuerst ausgeführt wird - und zwar direkt beim Deklarieren. Jede Klasse hat einen Konstruktor, ganz egal, ob dieser definiert wurde oder nicht. Weil er als erstes ausgeführt wird, initialisiert man üblicherweise alle Attribute und führt gegebenenfalls nötige Methoden aus - daher der Name "Konstruktor".

Wie eigentlich jede Funktion kann auch der Konstruktor überladen werden. Der Form wegen schreibt man auch meist gleich mehrere Konstruktoren. Allgemein unterscheidet man zwischen:

- Standardkonstruktor
- Konstruktor, der alle Attribute einzeln initialisiert (dafür gibt's keinen so tollen Fachbegriff)
- Copy- oder Kopier-Konstruktor

Allen Konstruktoren ist gemein, dass sie

1.) keinen - GAR KEINEN - Rückgabetyp haben. Auch das **void** ist Tabu.
2.) den selben Namen haben wie die Klasse, der sie angehören.

Der Standardkonstruktor

Was fällt dir ein, wenn du dieses Wort hörst?? Beim Standardkonstruktor handelt es sich um die Serviette, die man im Restaurant zum Essen dazu kriegt. Oder den Plastikbeutel, den man beim Einkaufen dazukriegt, sofern der nichts kostet. Der ist immer dabei. Auch die Klassen aus dem vorigen Kapitel haben immer einen gehabt.

Demzufolge wird er immer automatisch vom Compiler erzeugt. Das kannst du dir so vorstellen:

```
class House
{
   //...
};

House::House(void)
{
} //vom Compiler automatisch mit compiliert
```

Es steht zwar nichts drin, aber das ist dem Compiler egal. Denn das Programm braucht ihn beim Deklarieren eines Objektes immer, auch wenn er sich nur auf's Dasein beschränkt.

Im Gegensatz zu nicht-Objekten könntest du beim Objektinstanzieren zwei Schritte erkennen:

```
House A;
```
1.) Speicher für `A.m_Doors`, `A.m_Space` usw. reservieren
2.) Standardkonstruktor aufrufen

Folgende Deklaration macht das gleiche:

```
House B();
```

Wenn du also die Klammern wegläßt, kannst du dir sicher sein, dass der Standardkonstruktor aufgerufen wird.

Wer aber wirklich OOP machen will, füllt da noch einen sinnvollen Inhalt rein. Dazu muss man den Standardkonstruktor dann doch selbst geschrieben werden:

```
class House
{
   private:
      short m_Doors;
      short m_Windows;
      short m_Rooms;
      short m_Space;
      short m_Persons;
   public:
      //...
      House(void);
};

House::House(void)
{
   m_Doors = 0;
   m_Windows = 0;
   m_Rooms = 0;
   m_Space = 0;
   m_Persons = 0;
}
```

Der Konstruktor ist nun festgelegt und der Compiler brauch sich nicht mehr die Mühe machen. Das Objekt `House C;` würde also gleich mit allen Werten auf Null zur Welt kommen.

Kurz:

Der Standardkonstruktor ist eine parameterlose Funktion, die beim Instanzieren eines Objektes aufgerufen wird - ganz egal, ob man nun `House X;` oder `House X();` schreibt.

Normaler Konstruktor

Ein normaler Konstruktor ist alles, was einen oder mehrere Parameter hat. Ein typischer Konstruktor initialisiert gleich auf Anhieb alle Attribute:

```
class House
{
   private:
      short m_Doors;
      short m_Windows;
      short m_Rooms;
      short m_Space;
      short m_Persons;
   public:
      //...
      House(void);
      House(short D, short W, short R, short S, short P);
};

House::House(short D, short W, short R, short S, short P)
{
   m_Doors = D;
   m_Windows = W;
   m_Rooms = R;
   m_Space = S;
   m_Persons = P;
}
```

Aufruf des Konstruktors

Wie du diesen Konstruktor aufrufen kannst, ist klar:

```
House E(1, 2, 3, 4, 5);
```

Wenn du die Parameter mit Default-Werten belegst, kannst du dir dadurch ein bisschen Tipperei sparen. Wenn dadurch aber zwei Konstruktoren aneinander geraten, gibt der Compiler dir eine Fehlermeldung aus.

Nehmen wir an, für jeden der fünf Parameter gibst du einen Defaultwert an - der Compiler kann nicht entscheiden, ob er nun den Standardkonstruktor oder den mit den fünf Parametern nehmen soll.

Ein guter Kompromiss ist, ALLE Parameter mit Argumenten zu füttern und auf Defaultparameter gänzlich zu verzichten!

Auch gleich ganz nützlich ist es, dynamischen Speicher zu reservieren. Dann ist aber auch ein Destruktor erforderlich, um den Speicher wieder freizugeben (wenn der noch nicht befreit ist). Die Klasse `House` soll bei dieser Gelegenheit auch gleich einen dynamisch angelegten String erhalten, der den Strassennamen festhält.

```
class House
{
   private:
      //...
      char *m_Street;
   public:
      //...
      House(void);
      House(short D, short W, short R, short S, short P, const char
*Street);
};

House::House(void)
{
   m_Doors = 0;
```

```
   m_Windows = 0;
   m_Rooms = 0;
   m_Space = 0;
   m_Persons = 0;
   m_Street = new char[256];
}
```

`m_Street` muss unbedingt **delete**`[]` werden! Das kann der Destruktor übernehmen. Doch zunächst interressiert uns der...

Copy-Konstruktor

Dieser Konstruktor nimmt sämtliche Attribute aus einem Objekt, das als Parameter übergeben wurde, und kopiert sie in das Objekt des Konstruktors. Dadurch kannst du von einer exakten Kopie ausgehen.

```
class House
{
   //...
   public:
      //...
      House(void);
      House(const House &Obj);
      House(short D, short W, short R, short S, short P, const char
*Street);
};

House::House(const House &Obj)
{
   m_Doors = Obj.m_Doors;
   m_Windows = Obj.m_Windows;
   m_Rooms = Obj.m_Rooms;
   m_Space = Obj.m_Space;
   m_Persons = Obj.m_Persons;
   m_Street = new char[256];
   strcpy(m_Street, Obj.m_Street);
}
```

Das **const** ist empfehlenswert. Derjenige, der den Copy-Konstruktor mit

```
House D(E);
```

aufruft, kann sich sicher sein, dass `E` sich nicht ändern wird.

Der Destruktor

Um noch einmal auf den reservierten, aber nicht wieder freigegebenen Speicher, den `m_Street = new char[256];` reserviert hat, zurückzukommen: Irgendwie muss der ja auch wieder freigegeben werden! Eine Lösung mit einer normalen Methode ist unsauber, denn die könnte ja mehrmals ausgeführt werden.

Der Destruktor ist das Gegenstück zum Konstruktor. Er wird automatisch aufgerufen, wenn das Objekt seinen Gültigkeitsbereich verlässt. Auch hier gibt es eine Variante, die standardmäßig vom Compiler erstellt wird und nichts macht. Zweck ist es, Speicher auf dem Heap freizugeben oder Dateien zu schließen.

Weil er nicht willkürlich aufgerufen werden kann, erübrigen sich sowohl Rückgabetyp als auch Parameterliste (jegliche Möglichkeiten zur Überladung sind ebenfalls hin). Er ist mit

dem Klassennamen mit vorangestellter Tilde '~' ansprechbar. Für House nach neuestem Stand sähe er so aus:

```
class House
{
   private:
      //...
      char *m_Street;
   public:
      House(void); //Konstruktor
      ~House(); //Destruktor
}


House::House(void)
{
   m_Doors = 0;
   m_Windows = 0;
   m_Rooms = 0;
   m_Space = 0;
   m_Persons = 0;
   m_Street = new char[256];
}

House::~House()
{
   if(m_Street) delete[] Name;
   //m_Street = 0; //eigentlich unnötig
}
```

Aus lauter gutem Willen gewöhnst du dir selbstverständlich an, diese 4 (Default-Konstruktor, normaler Konstruktor, Copy-Konstruktor und Destruktor) jeder Klasse einzufügen. Auch wenn sie leer sind - dann geht's auch ohne externe Definition:

```
class House
{
   //...
   public:
      //...
      House(void) {}; //hier mal leer
      House(const House &Obj);
      House(short D, short W, short R, short S, short P, const char
*Street);
      ~House() {}; //hier mal leer
};
```

Die erste und die vierte Methode werden auch als leerer Konstruktor bzw. Destruktor bezeichnet. Geh einfach davon aus, dass m_Street hier nicht dynamisch angelegt wird.

explicit-Konstruktoren

Wenn du einen Konstruktor für eine Klasse geschrieben hast, der nur einen Parameter hat, kannst du ein Objekt bei einem Funktionsaufruf instanzieren, ohne den Konstruktor ausdrücklich aufzurufen. Dieses Argument erzeugt ein Objekt mit dem Namen des Parameters. Hier erstmal die Vorgeschichte:

```
class ExplClass
{
   private:
      int m_A;
      int m_B;
   public:
```

```
      ExplClass(int f)
      {
         m_A = f;
         m_B = f;
      }
      void PrintValues(void)
      {
         cout << m_A << "\n" << m_B << "\n";
      }
};

int main(void)
{
   ExplClass ABC(50);
   //...
}
```

ABC wird wie gewohnt instanziert.

Der Konstruktor verlangt nur einen Parameter. Dann brauchen wir noch eine Funktion, die als Parameter ein Objekt der Klasse ExplClass verlangt:

```
void Function(ExplClass DEF) //normale Funktion
{
   DEF.PrintfValues();
}

int main(void)
{
   ExplClass ABC(50);

   Function(17); //automatischer Konstruktoraufruf

   //...
}
```

In der Funktion ist dann immer das Objekt DEF frisch erschaffen worden. Das kann manchmal gewollt und manchmal ungewollt sein. Für den Fall, dass es ungewollt ist, gibt es das Schlüsselwort **explicit**, das direkt vor den Konstruktor geschrieben wird. Die automatische Umwandlung wird somit verboten; jeder Versuch wird mit einem Compilerfehler abgefangen. Das ermöglicht zusätzliche Kontrolle für dich.

Beispiel explicit1.

Auch folgendes ist mit einem Konstruktor möglich, der nur einen Parameter verlangt:

```
ExplClass DEF = 9;
```

Der Zuweisungsoperator dient hier dem Zweck, den geeigneten Konstruktor aufzurufen. Es ist dasselbe wie

```
ExplClass DEF(9);
```

Das zeigt sich auch im Beispielprojekt. Am besten, du probierst darin ein bisschen rum.

static-Elemente

static - das haben wir vorher dafür verwendet, um klarzumachen, dass eine Variable nur in einem Sourcefile verfügbar ist. In OOP kiommt ihm eine neue Bedeutung zu: Ein mit **static** deklariertes Attribut verhält sich wie eine Union, es gibt also nur eine Speicherstelle für die Variable in der ganzen Klasse:

```
class House
{
   private:
      //...
      static char m_Date[256]; //Datum, an dem die Häuser im
Speicher stehen
};

char *House::m_Date = "1.1.2003"; //<- in die Implementierungsdatei
rein
```

In der letzten Zeile wird die Variable initialisiert (deswegen das **char*** noch mal). Weil sie bei allen Instanzen der Klasse gleich ist, sollte sie nicht im Konstruktor initialisiert werden. Jedes Mal, wenn ein neues Objekt instanziert wird, würde die Variable überschrieben werden.

Wenn `m_Date` **public** wäre, könnte man so von außen auf die Variable zugreifen:

```
cout << House::m_Date;
```

Da dem aber nicht so ist, müssen wir uns was anderes überlegen. Man könnte Set- und Get-Methoden schreiben.

static-Methoden

Eine als **static** deklarierte Methode gibt es genau wie ein **static**-Element nur ein mal in der Klasse. Die sollte dann eine gemeinnützige Arbeit verrichten. Zum Beispiel ist für alle Objekte der Klasse `House` das Attribut `m_Date` gleich; da liegt es nahe, dafür allgemeinnützige Set- und Get-Methoden zu implementieren:

```
class House
{
   private:
      //...
      static char m_Date[256];
   public:
      static void SetDate(char *Date);
      static char* GetDate(void);
      //...
};
```

Zugriff erfolgt wie bei den static-Elementen mit dem Klassennamen und den Doppelpunkten `::` :

```
House::SetDate("2004");
```

Beispiel house2.

const-Methoden

Es gibt viele Methoden, in denen die Attribute einer Klasse nicht verändert werden. Bisher gab es in der `House`-Klasse nur die Methoden `Extend()` und `Set...()`, die in die Attribute geschrieben haben. Bei `Get...()` und `Statistics()` wurde nur aus den Attributen gelesen.

Damit du das dem Compiler ausdrücklich klarmachen kannst, schreibst du einfach das Schlüsselwort **const** hinter die Parameterliste der gewünschten Methode. So kann kein Klasseneigenes Attribut verändert werden. Auch eine Änderung über den Umweg einer Funktion ist nicht möglich, was einen großen Vorteil mit sich bringt.

Wenn versucht wird, ein Attribut in einer Funktion zu ändern, die von der **const**-Methode aufgerufen wird, bekommst du eine Fehlermeldung und weißt bescheid über eine garantiert ungewollte Handlung.

In House kann jede Get-Methode als **const** markiert werden:

```
class House
{
   //...
   public:
      House(void);
      House(const House &Obj);
      House(short D, short W, short R, short S, short P, const char
*Street);
      ~House(void);

      void Extend(short Rooms, short Space);
      void Statistics(void) const;

      short GetDoors(void) const;
      void SetDoors(short Doors);

      short GetWindows(void) const;
      void SetWindows(short Windows);
      //...
};
```

Damit weiß derjenige, der diese Klassendefinition zum ersten mal sieht, dass jegliche Umänderung der Attribute schon im Keim erstickt wird. Es wird außerdem sichergestellt, dass konstante Objekte auf diese Funktionen zurückgreifen können. Ohne das **const** hinter dem Methodenprototypen wäre im folgenden Fall die Statistikfunktion nicht ausführbar:

```
const House Hall;

Hall.Statistics();
```

Ein solches Objekt ist zwar eher ein Sonderfall (weil auch keine Set-Methoden gehen), aber trotzdem: Wo **const** drin ist, da sollte auch **const** draufstehen!

mutable

Aber auch bei **const**-Methoden gibt es Hintertüren: Einzelne Attribute, die auch in einer **const**-Methode geändert werden sollen, kannst du mit dem Schlüsselwort **mutable** versehen.

mutable führt dann wieder zu unsauberem Code, weil das **const**-Sein eines Objektes nicht missachtet werden sollte. Und wieder gibt es Spezialfälle, die **mutable** benötigen.

Beispiel const2.

inline-Methoden

Um dieses Schüsselwort nicht in Vergessenheit geraten zu lassen, füge ich es hier nochmal kurz ein. **inline** ist auch für Objektmethoden anwendbar. Es ist bei internen Definitionen sogar automatisch mit dabei.

Wenn eine Get-Methode aus House in der Klassendefinition mitdefiniert wird, würde es auf jeden Fall **inline** werden:

```
class House
{
   //...
   public:
      short House::GetDoors(void) {return m_Doors;}
      //...
};
```

Das Schlüsselwort this

In jedem Objekt steht ein Zeiger namens **this**, der auf das Objekt selbst zeigt, zur Verfügung. Er kann somit in allen Methoden verwendet werden - mit Ausnahme der **static**-Methoden.

Er kann dazu verwendet werden, um zu prüfen, ob eine übergebene Adresse die ist, die **this** enthält:

```
int House::IsThis(House *Obj)
{
   if(Obj == this) return 1;
   else return 0;
}
```

Attribute können ebenfalls über den Umweg dieses Zeigers angesprochen werden:

```
void House::SetDoors(short Doors)
{
   if(Doors >= 0) this->m_Doors = Doors;
}
```

Folgende Ausdrücke sind gleich:

```
this->m_Doors

(*this).m_Doors

m_Doors
```

Vielmehr ergibt sich ein Nutzen, wenn der Parameter in dieser Set-Methode den selben Namen hätte wie das Attribut. Hier nehmen wir mal an, das Attribut würde nicht nach einer Namensregel benannt sein und einfach Doors heißen:

```
void House::SetDoors(short Doors)
{
   if(Doors >= 0) this->Doors = Doors;
}
```

Zusammenfassung

Ich hoffe, du konntest in diesem Kapitel alles mit Erfolg gemeistert. Die ganzen Sachen mit OOP sind für viele Anfänger immer wieder eine kleine Hemmschwelle - genauso wie bei Zeigern. Für dieses Kapitel gilt ganz besonders: schau dir alle Beispielprojekte an und versteh sie. In vielen sind ein paar kleine Tipps, "wie man es macht", die im eigentlichen Tutorial gar nicht erwähnt werden (so syntaxmäßig).

Mit Konstruktoren ging es los. Sie übernehmen die Initialisierung der Attribute, denn ein Konstruktor wird immer dann aufgerufen, wenn ein Objekt deklariert wird. Du kannst dir soviele Konstruktoren wie du willst bauen; es gelten lediglich die Regeln des Überladens. Aufrufen kannst du sie dann mit geeigneten Parameterlisten.

Falls es nötig ist, ein Element auf dynamischen Speicher zu legen, ist der Konstruktor hervorragend dafür geeignet. Um die Freigabe kann sich dann der Destruktor kümmern. Den musst du allerdings speziell auf deine Attribute abrichten, sonst tut er nämlich nichts.

Im Anschluß sind die Wörter **this**, **explicit**, **mutable**, **static**, **inline** und **const** gefallen. Ganz wichtig ist es, Methoden, die die Attribute nicht ändern (direkt oder indirekt), als **const** zu deklarieren, um dem Compiler und dem davorsitzenden Programmierer einen Stein vom Herz zu nehmen.

Workshop

Fragen

1.) Darf man Konstruktoren in den **private**-Bereich stellen??
2.) Kann man es verbieten, Objekte einer Klasse zu erzeugen??
3.) **explicit** kann man nur auf Konstruktoren mit einem Parameter anwenden, richtig??
4.) Wieso erfindet man **mutable**, wenn man doch das **const**-Sein nicht umgehen sollte??
5.) Gibt es eine Alternative Lösung zu **static**-Membern??
6.) Was ist hieran falsch (ganz dumm gefragt)??

```
explicit mutable static register const unsigned int
EMSRCUInt;
```

7.) Um welche Attribute könnte man die Klasse `House` noch erweitern??

Programme

1.) Spendiere in deinen eigenen Beispielprojekten den Methoden je nach Bedarf die Zusätze **inline** und **const**!!
2.) Initialisiere deine Objektelemente mit einem Konstruktor (implementiere für jede deiner Klassen die drei Konstruktoren)!!

Links:

http://www.volkard.de/vcppkold/konstruktor.html

http://www.volkard.de/vcppkold/destruktor.html http://www.volkard.de/vcppkold/copykonstruktor.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop11.html
http://www.schornboeck.net/ckurs/ctor.htm

Kapitel 18

Dieses Kapitel soll dir die einzigartige Möglichkeit, Operatoren für deine Klassen anzupassen - also Überladen, beibringen. Jede Klasse hat ihre eigenen Attribute; der Compiler brauch deswegen ein bisschen Unterstützung.

Operatoren überladen

In C++ besteht für Klassen die Möglichkeit, bekannte Operatoren umzufunktionieren. Es gibt genügend Operatoren zum Überladen - neue zu erfinden wäre für den Compiler eine unlösbare Bürde; die Prioritäten der Operatoren ändern sich ebenfalls nicht. Wenn zum Beispiel sich zwei Häuser zu einem verbinden (die sind jetzt mal benachbart), müssten in ein neues Objekt die Daten zusammenaddiert werden, also:

```
House Klein1(2, 2, 2, 2, 2, "Hütte");
House Klein2(2, 2, 2, 2, 2, "Hütte");
House Mittel;
Mittel = Klein1 + Klein2;
```

Die letzte Anweisung ist leider noch unzulässig, weil eine Klasse kein einfacher Datentyp wie **int** oder **float** ist. Der arithmetische +-Operator ist dafür vorgesehen, einfache Datentypen (und daher keine Klassen) miteinander zu addieren.

Aber mit einem neuen Schlüsselwort können wir den Operator überladen. Überladen hast du schon im Zusammenhang mit Funktionen und Konstruktoren kennengelernt. Um das mit Operatoren zu erreichen, müssen wir eine Methode schreiben, in deren Namen das Schlüsselwort **operator** und der gewünschte umzufunktionierende Operator steht:

```
Rückgabetyp operator Zeichen (Parameter);
```

Für unsere House-Klasse sieht das so aus:

```
class House
{
   //...
   public:
      //...

      //Methode zum Fusionieren der Attribute zweier Häuser:
      House operator+ (House Obj);
};
```

operator+ ist der Methodenname. Jeder Operator kann mehrmals Überladen werden, aber jeweils mit eindeutigen Parametern! Es gibt einen Rückgabetyp, und der ist House. Die Definition für die Methode sieht so aus:

```
House House::operator+(const House Obj)
{
   House Sum;
   Sum.m_Doors = m_Doors + Obj.m_Doors;
   Sum.m_Windows = m_Windows + Obj.m_Windows;
   Sum.m_Rooms = m_Rooms + Obj.m_Rooms;
   Sum.m_Space = m_Space + Obj.m_Space;
   Sum.m_Persons = m_Persons + Obj.m_Persons;
   return Sum;
```

```
}
```

Beim Aufruf wird ein Objekt der Klasse `House` übergeben. In der Methode wird eine weitere Klasse instanziert. Die Attribute dieses Objektes setzen sich dann aus dem jeweiligen Attribut des Objektes, das die Methode aufruft, und Attribut des übergebenen `Obj`-Objektes zusammen. Zum Schluss wird das komplette Objekt zurückgegeben.

Den Operator "aufrufen"

Der Aufruf kann dann so erfolgen, wie man das schon seit eh und je macht:

```
klein1.operator+(klein2);
```

Um die Rückgabe des Objekts `Sum` nutzen zu können, kann diese Funktion als Zuweisung für ein Objekt erfolgen:

```
mittel = klein1.operator+(klein2);
```

Und jetzt bitteschön so wie wir es wünschen:

```
mittel = klein1 + klein2;
```

Diese Zeile und die darüber tun genau dasselbe. Wenn man nur diese Zuweisung sehen würde und nicht die Definition der Operator-Methode, würde man den ersten Aufruf leichter verstehen und es wäre gleich ersichtlich, dass es sich um einen überladenen Operator handelt.

> Wo hatten wir bloß schon überall einen überladenen Operator angewendet??
>
> Richtig, `cout` und `cin` lassen uns mit `<<` bzw. `>>` recht einfach Daten aus- und eingeben. Diese Objekte haben noch einige Operatoren im Gepäck. Die wirst du bei Zeiten noch kennenlernen.

Beispiel house3.

Alle überladbaren Operatoren

Mögliche Operatoren sind:

+	-	*	/
+=	-=	*=	/=
<<=	>>=	<=	>=
~	&=	^=	|=
++	--	%	&
<<	>>	%=	=
<	>	^	!
==	!=	|	&&
||	()	[]	->
new	**delete**		

Die lassen sich nicht alle gleich behandeln. Aber es gibt Gruppen:

Unäre Operatoren	+ - ~ ++ -- !
Binäre Operatoren	+ - * / += -= *= /= <<= >>= <= >= &= ^= |= % & << >> %= = < > ^ == != | && ||
Funktionelle Operatoren	() []
Sonderoperatoren	-> **new delete**

Alle diese könnte man so überladen, dass sie einem anderen Zweck dient; im Folgenden zeige ich jedoch die ursprünglich richtige Bedeutung.

Unäre Operatoren

Da + und - sowohl unär als auch binär vorkommen, müssen sich diese beiden Gruppen in ihren Parameterlisten unterscheiden. Unäre Operatoren gehorchen allgemein auf eine Überladung mit keinem Parameter:

```
Rückgabetyp operator Zeichen (void);
```

Für die Vorzeichen + und - wäre eine sinngemäße Überladung:

```
class Number
{
   private:
      long m_Value;
   public:
      void SetValue(long Value) {m_Value = Value;}
      long GetValue(void) {return m_Value;}

      Number() {m_Value = 0;}
      Number(long Value) {m_Value = Value;}
      Number(const Number *Obj) {m_Value = Obj->m_Value;}

      Number & operator+(void);
      Number & operator-(void);
};

Number & Number::operator+(void)
{
   return *this;
}

Number & Number::operator-(void)
{
   Number A(*this)
   A.m_Value = -m_Value;
   return A;
}
```

Für den unären bitweisen Negativ-Operator '~' gibt es folgende Implementation:

```
class Number
{
   //...
   public:
      //...
      Number & operator~(void);
```

```
};

Number & Number::operator~(void)
{
   Number A(*this);
   A.m_Value = ~m_Value;
   return A;
}
```

So geht das immer weiter. Hier noch die restlichen Operatoren überladen:

```
class Number
{
   //...
   public:
      //...
      Number & operator++(void);
      Number & operator--(void);
      bool operator!(void);
};

Number & Number::operator++(void)
{
   m_Value++;
   return *this;
}

Number & Number::operator--(void)
{
   m_Value--;
   return *this;
}

bool Number::operator!(void)
{
   if(m_Value == 0) return 0;
   return 1;
}
```

Für logische Operatoren empfehle ich **bool** als Rückgabetyp, bei den anderen (arithmetische und bitmanipulierende Operatoren) die Referenz auf das aufrufende Objekt.

Ein Beispielprojekt gibt's später, wenn alle Operatoren überladen wurden.

Binäre Operatoren

Welche waren das noch mal??

```
+ - * / += -= *= /= <<= >>= <= >= &= ^= |=
% & << >> %= = < > ^ == != | && ||
```

So. Da binäre operatoren immer zwei Ausdrücke benötigen, müssen die Überladungsfunktionen wohl einen Parameter bekommen.

```
Rückgabetyp operator Zeichen (Klasse & Objekt);
```

Nun, das arithmetische Plus hast du oben schon bei der Klasse `House` gesehen. Hier die Operatoren überladen:

```
class Number
{
   //...
   public:
      //...
      //arithmetisch
      Number operator+(const Number Obj);
      Number operator-(const Number Obj);
      Number operator*(const Number Obj);
      Number operator/(const Number Obj);

      Number & operator+=(const Number Obj);
      Number & operator-=(const Number Obj);
      Number & operator*=(const Number Obj);
      Number & operator/=(const Number Obj);

      Number operator%(const Number Obj);
      Number & operator%=(const Number Obj);

      //bitweise
      Number operator<<(const Number Obj);
      Number operator>>(const Number Obj);
      Number operator&(const Number Obj);
      Number operator|(const Number Obj);
      Number operator^(const Number Obj);

      Number & operator<<=(const Number Obj);
      Number & operator>>=(const Number Obj);
      Number & operator&=(const Number Obj);
      Number & operator|=(const Number Obj);
      Number & operator^=(const Number Obj);

      //logisch
      bool operator<(Number Obj);
      bool operator>(Number Obj);
      bool operator<=(Number Obj);
      bool operator>=(Number Obj);
      bool operator==(Number Obj);
      bool operator!=(Number Obj);
};

//arithmetisch
Number Number::operator+(const Number Obj) {Number A(*this);
A.m_Value += Obj.m_Value; return A;}
Number Number::operator-(const Number Obj) {Number A(*this);
A.m_Value -= Obj.m_Value; return A;}
Number Number::operator*(const Number Obj) {Number A(*this);
A.m_Value *= Obj.m_Value; return A;}
Number Number::operator/(const Number Obj) {Number A(*this);
A.m_Value /= Obj.m_Value; return A;}

Number & Number::operator+=(const Number Obj) {m_Value +=
Obj.m_Value; return *this;}
Number & Number::operator-=(const Number Obj) {m_Value -=
Obj.m_Value; return *this;}
Number & Number::operator*=(const Number Obj) {m_Value *=
Obj.m_Value; return *this;}
Number & Number::operator/=(const Number Obj) {m_Value /=
Obj.m_Value; return *this;}

Number Number::operator%(const Number Obj) {m_Value %= Obj.m_Value;
return *this;}
Number & Number::operator%=(const Number Obj) {Number A(*this);
A.m_Value %= Obj.m_Value; return A;}

//bitweise
```

```
Number Number::operator<<(const Number Obj) {Number A(*this);
A.m_Value <<= Obj.m_Value; return A;}
Number Number::operator>>(const Number Obj) {Number A(*this);
A.m_Value >>= Obj.m_Value; return A;}
Number Number::operator&(const Number Obj) {Number A(*this);
A.m_Value &= Obj.m_Value; return A;}
Number Number::operator|(const Number Obj) {Number A(*this);
A.m_Value |= Obj.m_Value; return A;}
Number Number::operator^(const Number Obj) {Number A(*this);
A.m_Value ^= Obj.m_Value; return A;}

Number & Number::operator<<=(const Number Obj) {m_Value <<=
Obj.m_Value; return *this;}
Number & Number::operator>>=(const Number Obj) {m_Value >>=
Obj.m_Value; return *this;}
Number & Number::operator&=(const Number Obj) {m_Value &=
Obj.m_Value; return *this;}
Number & Number::operator|=(const Number Obj) {m_Value |=
Obj.m_Value; return *this;}
Number & Number::operator^=(const Number Obj) {m_Value ^=
Obj.m_Value; return *this;}

//logisch
bool Number::operator<(Number Obj) {if(m_Value < Obj.m_Value)
return true; return false;}
bool Number::operator>(Number Obj) {if(m_Value > Obj.m_Value)
return true; return false;}
bool Number::operator<=(Number Obj) {if(m_Value <= Obj.m_Value)
return true; return false;}
bool Number::operator>=(Number Obj) {if(m_Value >= Obj.m_Value)
return true; return false;}
bool Number::operator==(Number Obj) {if(m_Value == Obj.m_Value)
return true; return false;}
bool Number::operator!=(Number Obj) {if(m_Value != Obj.m_Value)
return true; return false;}
```

In einem Beispielprojekt weiter unten werden alle Methoden etwas besser formatiert.

Der Zuweisungsoperator, den ich hier ausgelassen habe, wird eigentlich immer automatisch überladen. Er wandelt hier und da eine Zuweisung an ein neues Objekt in einen Konstruktoraufruf um (was sich mit **explicit** vermeiden lässt). Allerdings sollte immer auch der überladen werden, da Strings zum Beispiel nicht richtig kopiert werden:

```
class Number
{
   //...
   public:
      //...
      Number & operator=(const Number Obj);
};

Number & Number::operator=(const Number Obj)
{
   m_Value = Obj.m_Value;
   return *this;
}
```

House ist hier vielleicht einleuchtender:

```
class House
{
   //...
```

```
   public:
      //...
      House & operator=(House Obj);
};
```

operator+ ist der Methodenname. Jede **operator**-Methode ist einzigartig in einer Klasse, es kann also keine Überladung eines Operators mehrfach erfolgen. Es gibt einen Rückgabetyp, und der ist House. Die Definition für die Methode sieht so aus:

```
House & House::operator=(const House Obj)
{
   m_Doors = m_Doors + Obj.m_Doors;
   m_Windows = m_Windows + Obj.m_Windows;
   m_Rooms = m_Rooms + Obj.m_Rooms;
   m_Space = m_Space + Obj.m_Space;
   m_Persons = m_Persons + Obj.m_Persons;
   strcpy(m_Street, Obj.m_Street);
   return *this;
}
```

Funktionelle Operatoren

In Anwendung sehen die aus wie ein Funktionsaufruf bzw. ein Indexzugriff: `()` und `[]`.

Für Number und House gibt es keine sinnvollen Überladungen der funktionellen und speziellen Operatoren.

Der Funktionsaufrufoperator `()` sowie der andere (`[]`) haben folgende Parameterübernahme:

```
//Rückgabetyp operator Zeichen (Parameter)

class X
{
   private:
      //...
      int m_X[1000];
   public:
      //...
      X operator()(X Obj);
      int operator[](int Obj);
};

X X::operator()(X Obj)
{
   return TuEtwas(*this, Obj);
}

int X::operator[](int Obj)
{
   return m_X[Obj];
}
```

Dem Funktionsaufrufoperator kannst du auch mehrere Parameter übergeben, je nach Einsatzzweck. Wie auch immer, ein "Aufruf" würde so aussehen:

```
X i, a;

i(a);
int i2 = i[1];
```

Und nun das versprochene **Beispiel operator1**.

Zusammenfassung

Objektorientierte Programmierung bietet Programmierern eine heftige Wucht an Möglichkeiten, besonders die Operatorüberladung ist sehr wertvoll. Und sicher ist auch dieses Kapitel ziemlich schwierig gewesen.

Normalerweise versucht man, wenn man Operatoren überlädt, die Operationen, die mit den einfachen Datentypen möglich waren, auf den komplexen Datentyp zu übertragen. Dass das aber nicht immer so ist, beweist schon mal `cout`. Der Bitshift-Operator `<<` wird dazu verwendet, um Daten auszugeben.

Achte immer schön auf den Rückgabetyp! Bei binären Operatoren wie `+ - * /` sollte das Objekt selbst nicht verändert werden. Sehr wohl aber bei `+= -= *= /=` !

Workshop

gibt's diesmal nicht!

Links:

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop13.html
http://www.schornboeck.net/ckurs/operatoren.htm

Kapitel 19

Endlich kannst du auch in die Tiefen des objektorientierten Programmierens eindringen. Die Vererbung ist ein weiterer Schritt in Richtung Wiederverwendbarkeit von Code.

Vererbung

Ein Kind wird geboren - ein Junge. Sein Vater und sein Opa haben beide dunkle Haare und tiefblaue Augen. Wie sieht es nun mit dem Burschen aus?? Wird er auch dunkle Haare und blaue Äuglein bekommen, wenn seine Mutter "nichts dagegen hat"?? Gewiß wird er!!

Wie nennt man das?? Die Überschrift verät es!!

Diese Geschichte ist aber eher biologisch als informatisch. Die Grundidee ist jedoch die gleiche: Klassen können von anderen Klassen die Eigenschaften erben. Dieses Kapitel soll dir die Vererbung an einem einfachen Beispiel beibringen.

Etwas Theorie

Die Vererbung in OOP erlaubt soviele Elternteile wie man will - und genauso viele Nachkommen. Jeder Nachkomme hat alle Attribute und Funktionen der Eltern zur Verfügung, vorrausgesetzt, der Zugriffschutz ist richtig eingebaut. Abgeleitete Klassen - so heißen die Nachkommen in OOP wirklich - können Methoden der Basisklassen - der Eltern-Ausdruck in OOP - auch neu definieren, eben um der Erweiterung der Basisklasse gerecht zu werden.

Mit Zeigern ist auch noch weiteres möglich: Ein Zeiger auf eine Basisklasse kann ohne weiteres auf ein Objekt der abgeleiteten Klasse zeigen - auf JEDE abgeleitete Klasse. Das bezeichnet man Vielgestaltigkeit - Fachbegriff Polymorphie. So weit, so gut, kehren wir erst einmal wieder zur "simplen" Vererbung zurück:

Der Vater von House

Die altbekannte Klasse `House` soll in eine kleine Klassenhierarchie eingeordnet werden. Was für ein Vater wäre geeigneter als eine neue Klasse `Bauwerk` - oder um es ins Englische zu übertragen - `Building`?? Diese Klasse soll Informationen darüber enthalten, wann, wo, durch wen und womit das Bauwerk gebaut wurde.

```
class Building
{
   private:
      enum { StrLength = 256 };
      char *m_BuildDate;
      char *m_Place;
      char *m_Builder;
      char *m_Materials;
   public:
      Building();
      Building(const Building &Obj);
      Building(char *BuildDate, char *Place, char *Builder, char
*Materials);
      ~Building();
```

```
      Building &operator=(const Building &Obj);

      char *GetBuildDate(void) const { return m_BuildDate; }
      char *GetPlace(void) const { return m_Place; }
      char *GetBuilder(void) const { return m_Builder; }
      char *GetMaterials(void) const { return m_Materials; }
};
```

So, die Vorarbeiten sind getan. Die Strings sollen dynamisch angelegt werden und sich nach dem Konstruieren eines Objektes nicht mehr ändern lassen - deswegen keine Set-Methoden. Außerdem soll der Zuweisungsoperator überladen werden.

```
Building::Building()
{
   m_BuildDate = new char[StrLength];
   strcpy(m_BuildDate, "<unwichtig>");
   m_Place = new char[StrLength];
   strcpy(m_Place, "<unwichtig>");
   m_Builder = new char[StrLength];
   strcpy(m_Builder, "<unwichtig>");
   m_Materials = new char[StrLength];
   strcpy(m_Materials, "<unwichtig>");
}

Building::Building(const Building &Obj)
{
   m_BuildDate = new char[StrLength];
   strcpy(m_BuildDate, Obj.m_BuildDate);
   m_Place = new char[StrLength];
   strcpy(m_Place, Obj.m_Place);
   m_Builder = new char[StrLength];
   strcpy(m_Builder, Obj.m_Builder);
   m_Materials = new char[StrLength];
   strcpy(m_Materials, Obj.m_Materials);
}

Building::Building(char *BuildDate, char *Place, char *Builder,
char *Materials)
{
   m_BuildDate = new char[StrLength];
   strcpy(m_BuildDate, BuildDate);
   m_Place = new char[StrLength];
   strcpy(m_Place, Place);
   m_Builder = new char[StrLength];
   strcpy(m_Builder, Builder);
   m_Materials = new char[StrLength];
   strcpy(m_Materials, Materials);
}

Building::~Building()
{
   delete[] m_BuildDate;
   delete[] m_Place;
   delete[] m_Builder;
   delete[] m_Materials;
}

Building &Building::operator=(const Building &Obj)
{
   m_BuildDate = new char[StrLength];
   strcpy(m_BuildDate, Obj.m_BuildDate);
   m_Place = new char[StrLength];
   strcpy(m_Place, Obj.m_Place);
   m_Builder = new char[StrLength];
   strcpy(m_Builder, Obj.m_Builder);
   m_Materials = new char[StrLength];
```

```
    strcpy(m_Materials, Obj.m_Materials);
    return *this;
}
```

Ich weiß, das ist zunächst viel Code. Aber nun haben wir diese Klasse erst einmal hinter uns.

Ein Wort zu automatisch erzeugten Methoden

Auffällig ist zunächst, dass sich der Copy-Konstruktor und der überladene Zuweisungsoperator ziemlich ähnlich sehen oder eigentlich vollkommen gleich. Du hast in den vorigen Kapiteln erfahren, dass sowohl Copy-Konstruktor als auch Zuweisungsoperator automatisch vom Compiler überladen werden. Dieser Automatismus birgt aber einen großen Nachteil: er kann nicht richtig mit dynamischen Speicher umgehen!

Und wieso?? Einfach weil die automatisch erzeugten Methoden die Inhalte der Variablen einfach kopieren. Ein kopiertes Objekt - egal ob mit Zuweisung oder per Copy-Konstruktor - enthält genau die selben Werte in normalen Variablen - aber eben auch in Zeigern. Die Adressen werden kopiert und dynamisch angelegter Speicher bekommt gleich zwei Herscher verpasst. Wird das eine Objekt mittels Destruktor zerstört, so wird auch der Speicher freigegeben. Wenn es aber darum geht, auch noch das kopierte Objekt zu zerstören, hagelt es Windows-Fehlermeldungen - denn der nichtsahnende Compiler kann in solchen Fällen einfach nicht helfen.

Zwar ist es meist bequemer, etwas automatisch machen zu lassen, aber manuelle Arbeit kann dies meist nicht ersetzen. **Merk dir also, die automatisch erzeugten Methoden immer durch eigene abzulösen.** Besser noch ist die Regel der "Großen Drei" (Kopier-Konstruktor, Zuweisungsoperator und Destruktor): **Wenn man einen selber schreibt, sollte man die anderen 2 auch schreiben!**

Doch nun wieder zurück zur Vererbung! `House` und `Building` sollen eine Beziehung haben. Ein Haus ist schließlich auch ein Bauwerk! Ein Haus kann auch einen Ort haben, an dem es steht, es kann auch am Datum X gebaut worden sein usw. Aber Moment mal - wieso stehen diese Daten nicht höchstpersönlich in `House` drin??

Ganz einfach - diese Daten sind für alle Arten von Bauwerken von Bedeutung. OOP ist dafür konzipiert worden, um Code wiederzuverwenden. Es werden später noch die Klassen `Bruecke` und `Mauer` hinzukommen - englisch `Bridge` und `Wall`. Die können den `Building`-Code gleich mitverwenden.

Eine Klasse zum Erben machen

Eine Basisklasse braucht keine besonderen Merkmale aufzuweisen. Die abgeleitete Klasse dagegen muss mit einer Liste versehen werden, die ihre Basisklassen enthält:

```
class House : public Building
{
    //...
};
```

Hätte `House` gleich zwei Basisklassen, könnte es so aussehen:

```
class House : public Building, public Position
{
    //...
};
```

Das **public**, was da in der Liste auftaucht, gibt die Art der Vererbung an. Die ist zunächst einmal **public**, bis wir alle Arten kennen. Jetzt dreht es sich nochmal kurz um die Arten, die in der Klasse selber verwendet werden können:

Leider ist es mit diesen Klassenkonstruktionen nicht möglich, mit einer `House`-Methode direkt auf die privaten Elemente von `Building` zuzugreifen. Stattdessen muss ein Umweg über die Get-Methoden der `Building`-Klasse gemacht werden. In C++ ausgedrückt:

```
class House : public Building
{
   //...
   public:
      void Statistics(void)
      //...
};

void House::Statistics(void)
{
   //alter Code:
   cout << "Das Haus bei \"" << m_Street << "\" hat "<< m_Doors <<
" Tueren\n"
        << m_Windows << " Fenster\n"
        << m_Rooms << " Raeume\n"
        << m_Space << " Kubikmeter Wohnraum\nund es sind gerade "
        << m_Persons << " Leute drin.\n";

   //neuer Code:
   cout << "Es wurde anno " << GetBuildDate()
        << "\nvon " << GetBuilder()
        << "\nbei " << GetPlace()
        << "\nmit " << GetMaterials() << " gebaut!\n";

   //cout << m_Place; //führt zu Compilerfehler!!
}
```

Das führt zu einem hässlichen Mischmasch aus direktem (`m_Windows` ...) und indirektem Zugriff (`GetPlace()` ...). Den Kompromiss kann man eingehen, indem man **private**-Elemente zu **protected**-Elementen macht. **protected** ist die dritte Möglichkeit, Elemente unter Schutz zu stellen. Wobei es aber eher ein Mittelding zwischen **public** und **private** ist.

Ein Element, das als **protected** deklariert wurde, kann problemlos in den Methoden der abgeleiteten Klasse verwendet werden. Öffentlich verfügbar wie **public**-Elemente ist es aber keineswegs. Damit ist in `Statistics()` folgender Code möglich:

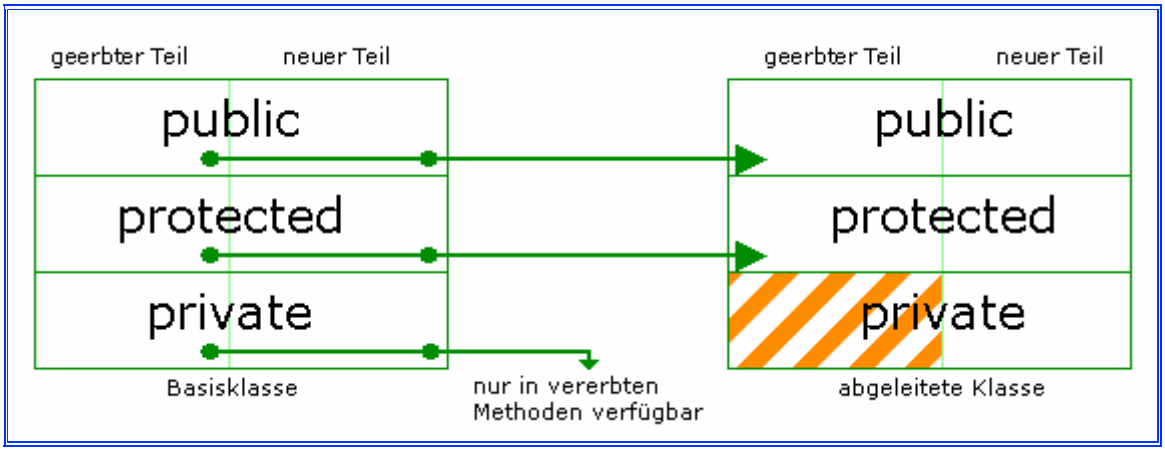
```
void House::Statistics(void)
{
   cout << "Das Haus bei \"" << m_Street << "\" hat "<< m_Doors <<
" Tueren\n"
        << m_Windows << " Fenster\n"
        << m_Rooms << " Raeume\n"
        << m_Space << " Kubikmeter Wohnraum\nund es sind gerade "
        << m_Persons << " Leute drin."
        << "\nEs wurde anno " << m_BuildDate
        << "\nvon " << m_Builder
        << "\nbei " << m_Place
        << "\nmit " << m_Materials << " gebaut!\n";
}
```

Speziell für diese Methode wirst du weiter unten noch eine elegantere Möglichkeit vorfinden, um die statistischen Daten von `Building` in der `Statistics()`-Funktion von `House` unterzubringen. `Building` wird seine eigene `Statistics()` bekommen.

Damit sind alle Arten geklärt. Bleibt nur noch zu klären, was diese drei p's bei der Vererbung bewirken:

public-Vererbung

Die oben gezeigte **`public`**-Vererbung ist die Art mit dem geringstem Zugriffschutz. Alle Elemente der Basisklasse bleiben auch in der abgeleiteten Klasse so, wie sie waren (**`private`**-Elemente werden vererbt, sind aber nur für Methoden der Basisklassen direkt verfügbar).



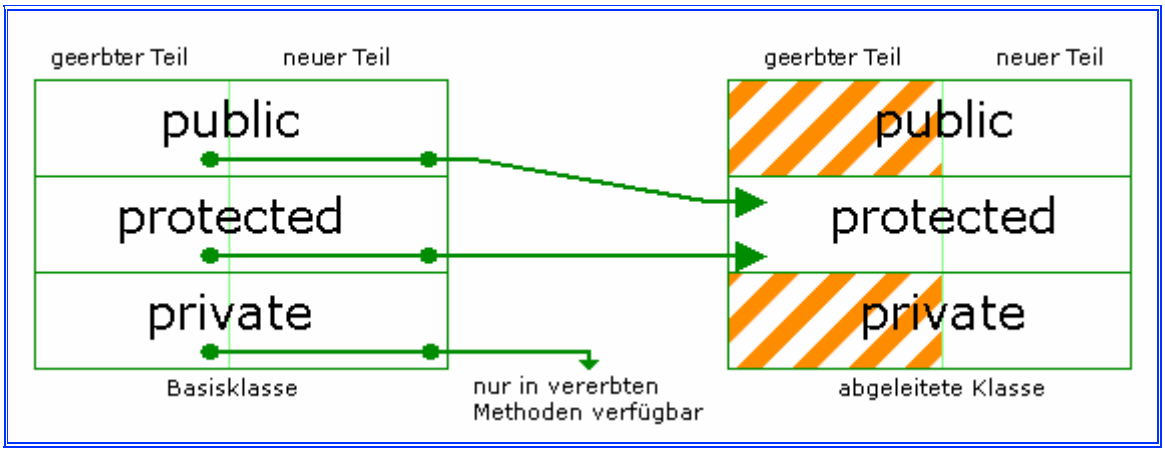
Diese Art der Vererbung zeigt das **Beispiel house4**.

protected-Vererbung

sieht so aus:

```
class House : protected Building
{
   //...
};
```

Hier ist das minimale Schutzniveau **`protected`**; kein vererbtes Element ist mehr **`public`** und alles wird zu **`protected`** umgewandelt (**`private`**-Elemente sind ebenfalls nur in den Methoden der Basisklasse direkt zu gebrauchen):

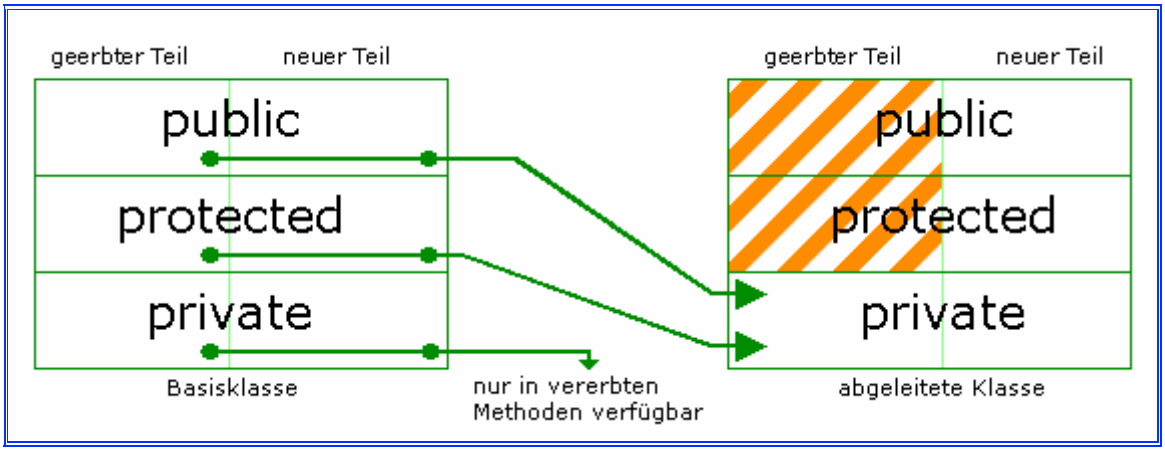


Auf Member der Basisklasse kannst du von Objekten der abgeleiteten Klasse nicht mehr von außen zugreifen. Mit neuen **public**-Methoden kannst du jedoch diese geschützten Methoden aufrufen und eventuell die Attribute ändern.

private-Vererbung

```
class House : private Building
{
   //...
};
```

Private Elemente aus der Basisklasse werden auch hier nicht für neue Methoden einsetzbar sein, aber der Rest wird **private** gemacht:



Die neue Klasse sollte dann neue öffentliche Methoden bereithalten, um die geschützten Elemente öffentlich im Programm einsetzen zu können. Das erfordert einigen Aufwand, ist aber ganz im Sinne der OOP. Der Zugriffschutz und die damit verbundene Kontrolle nehmen zu.

Du solltest dich einmal mit diesen Vererbungsarten auseinandersetzen und all jene Sachen implementieren, die dir bis jetzt mit der Vererbung möglich sind. Viele Tutorials schwören auf Einfachheit mit Klassen wie A und B oder B(Base) und D(Derived); wahrscheinlich sind diese zunächst besser geeignet als House und Building. Worauf ich hinaus will, ist, dass du dich jetzt mit dem "Wie (wird vererbt)" befasst.

Die Initialisierungsliste

Ok, die Klassen sind nun einander Erben:

```
class Building
{
   //...
};

class House : public Building
{
   //...
};
```

Beide Klassen haben ihre Konstruktoren. Es wäre nun Quatsch, in die Konstruktoren von `House` den Konstruktor-Code der Basisklasse `Building` einzubauen. Da bedient man sich eines weiteren Mittels, um das Kopieren des Codes zu ersparen - die Initialisierungsliste.

In der Klassendeklaration kommt ein Doppelpunkt nach der Parameterliste eines Konstruktors und dann kann es auch schon ordentlich losgehen mit dem Initialisieren. Du kannst einen Konstruktor der Basisklasse aufrufen (um den richtigen Konstruktor zu erwischen, gibst du entsprechend Parameter an; die können sogar direkt aus der Parameterliste des eigentlich aufzurufenden Konstruktors stammen) oder Attribute aus einfachen Datentypen mit Werten belegen.

```
class House
{
   //...
   public:
      //...
      House(void) : Building();
      House(const House &Obj) : Building(Obj);
      House(short D, short W, short R, short S, short P,
                const char *Street) : Building();

      House(short D, short W, short R, short S, short P, const char
*Street,
                char *BuildDate, char *Place, char *Builder,
                char *Materials) : Building (BuildDate, Place,
Builder, Materials);
};

//Definiert wird später!!!
```

Schon klar, es sieht jetzt ein bisschen bekloppt aus. Im Editor aber schaut es besser aus. Dem letzten Konstruktor könnte man kürzere Parameternamen verpassen...

Eine Initialisierungsliste ist auch die einzige Möglichkeit, um **`const`**-Attribute mit Werten zu belegen. Im Konstruktor selbst ist es nicht möglich. Andere Konstanten, die speziell für eine Klasse gelten, müssen mit **`enum`** erzeugt werden. Kleiner Beispielcode:

```
class X
{
   private:
      enum { StrLength = 1392, MaxValue = 4500 };
      const int m_Number;
      const char m_String[StrLength];
      int m_Y;
   public:
      X() : m_Number(100), m_String("Klasse X"), m_Y(50) {}
};
```

Die Initialisierungsliste ist, wie du schon gesehen hast, nicht zwingend in einer Klassenkonstruktion unterzubringen. Lässt du sie weg, werden die Default-Konstruktoren der Basisklassen aufgerufen. Eine abgeleitete Klasse ist nicht gezwungen, die

Initialisierungsliste mit all ihren Vorfahren-Konstruktoren zu füllen; immer wenn ein Konstruktor mit Initialisierungsliste aufgerufen wird, wird die Initialisierungsliste vor dem eigentlichen Konstruktor in der angegebenen Reihenfolge abgearbeitet.

Die Hintertür bei der Vererbung

Für den äußersten Notfall (z.B. wenn du unsauberen Code schreiben willst) kannst du Elemente in einer abgeleiteten Klasse unter beliebigen Zugriffschutz stellen und damit klarstellen, dass dir egal ist, wie das Element im Regelfall vererbt werden würde.

```
class A
{
   protected:
      int m_1;
      int m_2;
      void Function(void) {m_1 = 1; m_2 = 2;}
};

class B : private A
{
   public:
      using A::Function;
};
```

Nun ist alles aus `A` in `B` **`private`**, außer `Function()`. Diese Methode wird zum **`public`**-Bereich portiert. Auf die könntest du wie auf eine normale Methode zugreifen.

Mit **`using`** `<Klasse>::<Element>;` wird das Element in einen anderen Bereich "gezogen". Wie wohl zu sehen ist, brauchst du keine Parameterlisten anzugeben.

Wenn mehrere Elemente die gleichen Namen haben

Wenn nicht nur in einer Basisklasse und einer daraus abgeleiteten Klasse gleichnamige Elemente deklariert sind, kann es sein, dass der Compiler nicht richtig entscheiden kann, welches Element er nehmen soll. Hier hilft wieder der Bereichsauswahloperator `::` :

```
class Base1
{
   public:
      int m_1;
};

class Base2
{
   public:
      int m_1;
      void F1(void);
};

class Derived : public Base1, public Base2
{
   public:
      int m_1;
      void F1(void);
};

//beim Aufruf:

Derived Obj;
```

```
int main(void)
{
   Obj.F1();               //von Abgeleitet
   Obj.Base2::F1();        //von Basis2
   Obj.Var1 = 20;          //von Abgeleitet
   Obj.Base1::Var1 = 100;  //von Basis1
   //Obj.Base1::F1();        //Base1 hat keine F1()!
   //...
}
```

Doch auch sowas ist wieder eine Notlösung. Wenn es möglich ist, sowas bitte vermeiden.

Zusammenfassung

Dies war wieder mal so ein Hammer-Kapitel. Vererbung ist hart, da muss man durch!

Du hast die Basisklasse von `House` kennengelernt. Ich hätte die Daten aus der `Building`-Klasse auch in die `House`-Klasse reinstopfen können. Nur dann wäre das ja keine Vererbung und auch nicht sonderlich OOP-mäßig.

Ein Haus ist ein Bauwerk - ohne Frage. Ein Bauwerk ist aber nicht unbedingt immer ein Haus. Im folgenden Kapitel wirst du weitere Klassen kennenlernen, die auch von der `Building`-Klasse profitieren. Wie du in **house4** gesehen hast, könnte man Bauwerke wie die Freiheitsstatue oder Denkmäler von `Building` instanzieren.

Sehr bedeutsam ist die Art und Weise, was die verschiedenen Zugriffsschutz-Niveaus von C++ beim Vererben bewirken. Kurz und bündig hier ein kleines Vererbungsschema:

Basisklasse\Vererbung	**public**	**protected**	**private**
public	**public**	**protected**	**private**
protected	**protected**	**protected**	**private**
private	-	-	-

Die Initialisierungsliste ist dazu da, um die richtigen Konstruktoren aufzurufen. So müssen die `House`-Konstruktoren keinen Code von den `Building`-Konstruktoren enthalten. Eine von `House` abgeleitete Klasse braucht in ihrer Initialisierungsliste nur die `House`-Konstruktoren aufzurufen; um die `Building`-Konstruktoren muss sie sich nicht kümmern.

Die Vererbungsgesetze reinlegen?? Geht mit **`using`** `<Klasse>::<Element>;`, wenn es im gewünschten Bereich steht. Ist aber nicht zu empfehlen.

Und nun noch ein paar Fragen.

Workshop

Fragen

1.) Was für Elemente enthält `House` jetzt??
2.) Welche Vererbungsart ist vorzuziehen??
3.) Welche Aufgabe hat **`protected`**??
4.) Was passiert mit **`private`**-Elementen bei der Vererbung??
5.) Kann man auch **`private`**-Elemente mit der **`using`**-Anweisung unter einen anderen Zugriffschutz stellen??
6.) Wieso kann man Konstanten nicht einfach im Konstruktor initialisieren

lassen??

Programme

1.) Schreibe ein Rundenspiel im objektorientierten Stil, oder setze GDO in OOP um!

Links:

http://www.volkard.de/vcppkold/vererbung.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop17.html
http://www.schornboeck.net/ckurs/vererbung.htm

Kapitel 20

Polymorphismus

Ein Zeiger der Basisklasse kann wie üblich auf Objekte der Basisklasse zeigen. Eine abgeleitete Klasse enthält aber auch die Elemente ihrer Basisklasse. Folglich sollte es doch funzen, mit einem Basisklassenzeiger auf ein Objekt einer abgeleiteten Klasse zu zeigen. Und das klappt tatsächlich!!

Das ist die Grundlage für Polymorphismus (oder auch Polymorphie genannt, wer's so lieber hat) - zu gut deutsch "Vielgestaltigkeit". Ein Objekt der abgeleiteten Klasse wird auf dem Heap dynamisch reserviert und der besagte Basisklassenzeiger nimmt die Adresse an. Wenn ein anderes Objekt an der Reihe sein soll, gibt man den Speicher frei und erzeugt ein neues Objekt einer anderen Klasse.

`Building` soll in diesem Kapitel zwei weitere "Kinder" bekommen. Ein mal `Wall` und das andere mal `Bridge` - jeweils **`public`**.

```
class Wall : public Building
{
   protected:
      float m_Length;
      float m_Width;
      float m_Height;
   public:
      float GetLength(void) const { return m_Length; }
      void SetLength(float Length) { m_Length = Length; }

      float GetWidth(void) const { return m_Width; }
      void SetWidth(float Width) { m_Width = Width; }

      float GetHeight(void) const { return m_Height; }
      void SetHeight(float Height) { m_Height = Height; }

      Wall(void) { m_Length = 0; m_Width = 0; m_Height = 0; }
      Wall(float L, float W, float H) { m_Length = L; m_Width = W;
m_Height = H; }
      Wall(const Wall &Obj)
      {
         m_Length = Obj.m_Length;
         m_Width = Obj.m_Width;
         m_Height = Obj.m_Height;
      }
      ~Wall(){}

      Wall &operator=(const Wall &Obj)
      {
         m_Length = Obj.m_Length;
         m_Width = Obj.m_Width;
         m_Height = Obj.m_Height;
         return *this;
      }
};

class Bridge : public Building
{
   protected:
      float m_Length;
      float m_Depth;
```

```
   public:
      float GetLength(void) const { return m_Length; }
      void SetLength(float Length) { m_Length = Length; }

      float GetDepth(void) const { return m_Depth; }
      void SetDepth(float Depth) { m_Depth = Depth; }

      Bridge(void) { m_Length = 0; m_Depth = 0; }
      Bridge(float L, float D) { m_Length = L; m_Depth = D; }
      Bridge(const Bridge &Obj)
      {
         m_Length = Obj.m_Length;
         m_Depth = Obj.m_Depth;
      }
      ~Bridge(){}

      Bridge &operator=(const Bridge &Obj)
      {
         m_Length = Obj.m_Length;
         m_Depth = Obj.m_Depth;
         return *this;
      }
};
```

Die beiden Klassen sind ziemlich einfach gehalten. Jetzt soll es aber auch gleich mal losgehen mit dem Polymorphismus:

```
Building *pBuilding = new Building;
delete pBuilding;

pBuilding = new Wall(1.0f, 2.0f, 3.0f);
delete pBuilding;

srand(static_cast<unsigned int>(time(NULL)));
switch(rand()%3)
{
   case 0: pBuilding = new House;
           break;

   case 1: pBuilding = new Wall;
           break;

   case 2: pBuilding = new Bridge;
   default:break;
}
delete pBuilding;
```

Weil jede der verwendeten Klassen einen `Building`-Teil hat, kann der `pBuilding`-Zeiger auf diesen Teil mit dem `->`-Operator zugreifen. Und wie sieht's dagegen bei den hinzugekommenen Methoden aus??

So wie es jetzt konstruiert ist, können wir die Methoden z.B. von `House` nicht aufrufen. Eben weil `Building` diese Methoden nicht kennt. Deswegen gibt es sogenannte

Virtuelle Methoden

Damit wir trotzdem noch auf `Extend()` und Co von `House` zugreifen können, müssen wir diese Methoden in `Building` als virtuelle Methoden bekanntgeben. **`virtual`** ist das Schlüsselwort, ein Zusatz für Methoden.

```
class Building
{
   //...
   public:
      //...
      virtual void Statistics(void)
      {
         //sähe so aus:
         cout << "Es wurde anno " << m_BuildDate
              << "\nvon " << m_Builder
              << "\nbei " << m_GetPlace
              << "\nmit " << m_GetMaterials << " gebaut!\n";
      }
};
```

Code in dieser Funktion ist optional. Jetzt hat `Building` richtig offiziell eine `Statistics`-Funktion.

```
Building *pBuilding = new Building;
pBuilding->Statistics();       //ruft Building::Statistics() auf
delete pBuilding;

pBuilding = new House;
pBuilding->Statistics();       //ruft House::Statistics() auf
delete pBuilding;
```

Beidesmal der selbe Code, die Auswirkungen sind aber unterschiedlich!

Das Schlüsselwort **`virtual`** gibt dem Compiler bekannt, dass es eine gleichnamige Funktion auch in mindestens einer der abgeleiteten Klassen geben kann. Der Compiler schaut zuerst nach, ob die Klasse `House` eine Funktion `Statistics()` hat. Wenn ja, wird diese ausgeführt, ansonsten (wie in den Fällen `Wall` und `Bridge`) die virtuelle, gleichnamige Methode in `Building`.

Virtuelle Destruktoren

Das Beispiel grad eben war eigentlich nicht ganz korrekt:

```
Building *pBuilding = new House;
pBuilding->Statistics();
delete pBuilding;
```

Bisher haben nur Building und House Elemente, die auf dynamischen Speicher basieren. Aber auch schon da ist ein virtueller Destruktor wichtig.

Mit **`delete`** `pBuilding;` wird nämlich gar nicht der Destruktor von `House` aufgerufen, sondern der von `Building`. Das reicht natürlich nicht aus. Und jetzt kommt, was kommen muss - der virtuelle Destruktor.

```
class Building
{
   //...
   public:
      //...
      virtual ~Building()
      {
         //...
      }
};
```

Dadurch entstehen keine Speicherlöcher mehr, die ein `House`-Objekt zurücklässt. Der `Building`-Destruktor wird danach ausgeführt.

Übrigens: Wenn eine Methode in einer Klasse als **`virtual`** deklariert wird, ist sie in den abgeleiteten Klassen auch virtuell. Du liegst nicht falsch, wenn du jede Methode, die nach dieser Regel sowieso virtuell ist, mit einem **`virtual`** versiehst.

Abstrakte Basisklassen

Da wir von `Building` verschiedene Klassen abgeleitet haben, von denen wir Objekte instanzieren können, brauchen wir ab sofort keine Objekte von `Building` mehr (gut, die Miss Liberty aus dem vorigen Kapitel; aber dafür könnte man eine neue Klasse entwerfen).

Es gibt zwei Wege, um zuverlässig zu verbieten, Objekte von Klassen zu instanzieren. Nummero 1 - die Konstruktoren in den **`private`**-Bereich zu schieben - das hat aber mit Vererbung gar nicht so viel zu tun. Nummero 2 - die Klasse zu einer abstrakten Basisklasse zu machen.

Dazu müssen wir nur eine virtuelle Methode mit Null "initialisieren". Zeiger der Basisklasse kann man selbstverständlich noch erstellen.

```
class Building
{
   //...
   public:
      //...
      virtual void Extend(void) = 0;
};
```

Code für diese Methode für diese Klasse ist dadurch nicht mehr möglich. Falls du ein Objekt von dieser Klasse erstellen willst, wirst du eine Fehlermeldung bekommen, denn das Erstellen ist ja verboten.

Ein Nachteil bei dieser Abstraktion ist, dass jede abgeleitete Klasse eine Neudefinition dieser virtuellen Funktion vornehmen muss, auch wenn diese keinen Inhalt hat. Eine solche virtuelle Methode wird auch als rein virtuelle Methode bezeichnet.

Beispiel house5.

Zusammenfassung + Workshop

Heut nicht!

Links:

http://www.volkard.de/vcppkold/polymorphie.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop17.html
http://www.schornboeck.net/ckurs/virtual.htm

Kapitel 21

Templates sind in C++ Konstrukte, die Definitionen unabhängig von den Datentypen erlauben. Kommt dir bekannt vor?? Funktionen überladen und Makros - ja die gibt es schon. Hereinspaziert zu einem Kapitel (oder zwei) zur generischen Programmierung!

Templatefunktionen

Wenn du eine Funktion für mehrere Datentypen verfügbar machen willst, kannst du sie entweder überladen, oder aber zu einer Templatefunktion machen. Der Vorteil einer Templatefunktion ist, dass der Funktionsrumpf nicht mehrere Male mit datentypspezifischen Abwandlungen geschrieben werden muss, was sich besonders bei umfangreichen Funktionen bemerkbar macht. In gewisser Weise kannst du dir eine Templatefunktion wie ein Makro vorstellen.

Damit der Compiler weiß, dass es sich um eine Templatefunktion handelt, muss ihm das natürlich auf eine besondere Art klargemacht werden. Deklaration, Definition und Aufruf sieht nicht wie bei einer normalen Funktion aus. So könnte eine Deklaration aussehen:

```
template <class gTyp> Funktionskopf;
```

oder:

```
template <typename gTyp> Funktionskopf;
```

template, **class** und **typename** sind C++-Schlüsselwörter. Von **template** (deutsch: Schablone) kommt der Name Templatefunktion. **class** kommt auch bei Klassen vor, hier hat es aber eine andere Bedeutung. **typename** kann anstelle von **class** verwendet werden (nicht bei Klassen, nur bei Templatefunktion).

class und **typename** stehen für den noch unbekannten Datentypen, der erst bei Aufruf dieser Funktion bekannt wird. `gTyp` gibt an, wie der unbekannte Datentyp in der Funktion heißen soll. **class** `gTyp` bzw. **typename** `gTyp` müssen in eckigen Klammern `<>` stehen. Funktionskopf gibt bekannt, welchen Rückgabetyp, welchen Namen und welche Parameter die Templatefunktion haben soll. Als Rückgabetyp und als Parameter kann (sollte) `gTyp` eingesetzt werden.

Typisches Beispiel für eine Deklaration:

```
template <class T>
T Algorithmus1 (T a, T b, T c);
```

Als generischer Typ (`gTyp` von oben) wird **class** `T` angegeben, wobei natürlich auch **typename** `T` möglich wäre. Als Rückgabetyp wird hier `T` verwendet. `Algorithmus1()` hat drei Parameter, jeweils vom Typ `T`.

Nun kommt die Definition der Templatefunktion. `Algorithmus1()` soll zur Einfachheit die Summe der drei Parameter zurückgeben:

```
template <typename T>
T Algorithmus1 (T a, T b, T c)
{
```

```
    return a+b+c;
}
```

Du kannst, wie du es gewohnt bist, auch neue Variablen in der Funktion erschaffen. Bei Templatefunktion nimmst du dazu typischerweise den generischen Typen:

```
template <class T>
T Algorithmus1 (T a, T b, T c)
{
    T d = a+b+c;
    return d;
}
```

Beide Definitionen machen vom Ergebnis her dasselbe. Die zweite Funktion gebraucht aber eine weitere Variable, um die Summe der drei Parameter zwischenzuspeichern.

Beim Aufruf einer Templatefunktion kann nach dem Namen der spezielle Typ, durch den der generische Typ ersetzt werden soll, in eckigen Klammern angegeben werden, gefolgt von der Parameterliste in normalen Klammern:

```
int main(void)
{
    int A = 5, B = -26, C = -48, D = 0;
    D = Algorithmus1<int>(A, B, C);
    cout << D; //-69

    return 0;
}
```

Beispiel template1.

Du kannst als generischen Typen auch einen anderen Typen angeben, als die Parameter eigentlich sind:

```
int main(void)
{
    long A = 5, B = -26, C = -48, D = 0;
    D = Algorithmus1<char>(A, B, C);
    cout << static_cast<long>(D); //-69
}
```

Wenn die Datentypen der Argumente gleichartig sind, kann die Angabe des speziellen Datentyps auch weggelassen werden. Der Compiler findet dann selbst den Typ heraus:

```
int main(void)
{
    int A = 5, B = -26, C = -48, D = 0;
    D = Algorithmus1 (A, B, C);
    cout << D; //-69

    return 0;
}
```

Die Zuweisung `D = Algorithmus1<int>(A, B, C);` macht also dasselbe wie `D = Algorithmus1(A, B, C);` . Sind die Datentypen der Argumente allerdings nicht gleich, musst du selbst angeben, welche Datentypen du haben willst.

Beispiel template2.

Templatefunktionen mit mehreren generischen Typen

Wenn mehr als ein generischer Typ für die Funktion verfügbar sein soll, kannst du diese in der Definition (und Deklaration) durch Kommas trennen:

```
template <class Parameter, class Returntype>
Returntype Funktion1 (Parameter a, Parameter b)
{
   a += b;
   a *= b;
   a -= b;
   a /= b;
   return static_cast<Returntype>(a);
}
```

Hier wird nach ein paar Rechnungen der Parameter `a` zum Datentyp `Returntype` gecastet. Da kein Parameter vom Typ `Returntype` ist, kann die Funktion auch nicht den richtigen Typ für `Returntype` herausfinden, die Liste der generischen Typen muss also beim Aufruf dieser Templatefunktion angegeben werden:

```
int main(void)
{
   int A = 43, B = 36;
   float X = 99.323f, Y = 43.345f;

   double Ergebnis = Funktion1<signed int, double>(A, B);
   cout << Ergebnis; //78

   float Ergebnis = Funktion1 (X, Y); //beide Typen sind eindeutig
float!
   cout << Ergebnis; //??

   return 0;
}
```

Beispiel template3.

Natürlich können auch normale Parametertypen in der Parameterliste stehen:

```
template <class Parameter>
void Funktion1 (int a, Parameter b);
```

Die Funktion verlangt als erstes Argument einen **`int`**, `b` ist hingegen ein generischer Typ.

Beispiel template4.

Zusammenfassung

Wenn du Probleme bei den Templatefunktionen hattest, solltest du bei diesen noch ein bisschen üben. Denn das folgende Kapitel ist noch komplizierter zu verstehen.

Templatefunktionen sind z.B. sinnvoll für Algorithmen und Formeln aus der Mathematik einzusetzen, oder generell überall da, wo es 2 oder mehr Überladungen des selben Vorgangs gibt.

Links:

http://www.volkard.de/vcppkold/templatefunktionen.html

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop16_1.html
http://www.schornboeck.net/ckurs/generisch.htm

Kapitel 22

Weiter geht es mit objektorientierter und generischer Programmierung. Templateklassen ermöglichen es, Klassen zu konstruieren, die sowohl Attribute als auch Methoden haben, die nicht an einen speziellen Typ gebunden sind.

Templateklassen

Es ist möglich Klassen mit generischen Typen zu erstellen, also generische Klassen. Diese können normale Datentypen und Methoden enthalten, aber auch generische Attribute und generische Methoden. Ein Beispiel zur Deklaration einer generischen Klasse:

```
template <class T>
class Data;
```

Generische Parameter können wie gewohnt mit **class** oder **typename** definiert werden. **class** wird anschließend für die Klassendeklaration verwendet (wie gesagt funktioniert das Schlüsselwort **typename** hier NICHT!).

```
template <class T>
class Data
{
   private:
      T Var1;
      T Var2;
      T Var3;
   public :
      Data (){}
      Data (T A, T B, T C)
      {
         Var1 = A;
         Var2 = B;
         Var3 = C;
      }
      void Swap(void);
      T Add(T Extra);
};
```

Zunächst wurden 3 **private**-Attribute vom generischen Typ deklariert, dann ein leerer Konstruktor und danach ein Initialisierungskonstruktor mit 3 generischen Parametern, der auch gleich definiert wurde. Bis dahin sollte dir alles klar sein.

Die Funktionen `Swap()` und `Add()` sollen dir zeigen, wie du klassenexterne Definitionen von Methoden vornehmen kannst. Zunächst zu `Swap()`:

```
template <class T>
void Data<T>::Swap (void)
{
   T buf = Var2;
   Var2 = Var1;
   Var1 = Var3;
   Var3 = buf;
}
```

Zunächst wieder **template** und die Typenliste in den eckigen Klammern, dann der Rückgabetyp der Methode. Es folgt der Klassenname, dahinter in eckigen Klammern die

Namen der generischen Typen (in der richtigen Reihenfolge). Danach kommt der Auswahloperator `::` und der Methodenname mit Parameterliste. Nun die Funktion `Add()`:

```
template <class T>
T Data<T>::Add (T Extra)
{
   return Var1 + Var2 + Var3 + Extra;
}
```

Der Syntax ist der Gleiche wie bei `Swap()`, außer dass `Add()` einen Rückgabetyp und einen Parameter hat. Eine solche Konstruktion ist sicher sehr verwirrend, vor allem, wenn es mehr generische Typen und Parameter gibt.

Jetzt brauchen wir nur noch ein Objekt dieser generischen Klasse. Bei der Instanzierung eines solchen generischen Objekts muss der Typ angegeben werden, und zwar wieder in eckigen Klammern:

```
Data <long double> Position; //Aufruf des leeren Konstruktors mit
long double
Data <short> Blabla (100, 200, 300); //Attribute sind short
Data <char*> Strings ("Hallo", ", ", "Welt!");//Attribute sind
Zeiger auf char
```

Wenn du nun Methoden dieser Klasse aufrufen oder Attribute ändern willst, musst du nicht noch einmal die Typen angeben.

Beispiel template5.

Nichtgenerische Template-Parameter

Im Beispiel Template5 hast du gesehen, dass in den eckigen Klammern nicht nur generische Typen sein können, sondern auch fest definierte Typen. `T` und `I` sind im gesamten Template verfügbar. Anstatt `class A` oder `typename A` könnte z.B. `int A` oder `bool A` stehen. Beim Aufruf einer Funktion oder beim instanzieren eines Objektes mit einem solchen Template-Parameter muss zwingend ein passendes Argument übergeben werden. Fest definierte Template-Parameter kannst du sinnvoll bei Klassen einsetzen (z.B. um Größen von Feldern zu bestimmen, siehe **template5**), weniger Sinn machen diese Template-Parameter bei Funktionen, da es für solche Zwecke schließlich die ganz normalen Parameter gibt. Nichtgenerische Template-Parameter kannst du außerdem weder in einer Funktion noch in einer Klasse bzw. deren Methoden verändern. Zudem dürfen sie keine Fließkommatypen (**float**, **double** und **long double**) sein und dürfen keine Variablen sein.

Spezialisierungen für Templates

In manchen Fällen kann es sein, dass ein Template zwar für einen Großteil der möglichen Typen gut funktioniert, es aber für einen bestimmten Typen eine andere, spezielle Lösung geben soll. Beispielsweise würde die Methode `Add()` mit `T = char*` bei ihrem Aufruf einen Fehler auslösen. Die Klasse `Data` kann man so spezialisieren, dass `Add()` nichts tut, oder Strings aneinander anhängt:

```
//ursprüngliche Klasse:
template <class T>
class Data
{
   private:
      T Var1;
      T Var2;
```

```
      T Var3;
   public :
      Data (){}
      Data (T A, T B, T C)
      {
         Var1 = A;
         Var2 = B;
         Var3 = C;
      }
      void Swap(void);
      T Add(T Extra);
};

//spezialisierte Klasse mit T = char*:
template <>
class Data <char*>
{
   private:
      char* Var1;
      char* Var2;
      char* Var3;
   public :
      Data (){}
      Data (char* A, char* B, char* C)
      {
         Var1 = A;
         Var2 = B;
         Var3 = C;
      }
      void Swap(void);
      char* Add(char* Extra);
};
```

Der generische Typ `T` kann in der **`char*`**-Spezialisierung von Data nun nicht mehr benutzt werden. Jedes Element einer Klasse (nicht Funktion) muss in der Templateklasse sowie in der spezialisierten Templateklasse enthalten sein, auch wenn einige Elemente nicht gebraucht werden.

Hier die Definitionen für `Swap()` und `Add()`:

```
#include <cstring>

//...

template <>
void Data<char*>::Swap (void)
{
   char Buf[1000];
   strcpy(Buf, Var2);
   strcpy(Var2, Var1);
   strcpy(Var1, Var3);
   strcpy(Var3, Buf);
}

template <>
char* Data<char*>::Add (char* Extra)
{
   char* Str = strcat(Var1, Var2); //Var2 wird an Var1 angehängt
   Str = strcat(Str, Var3);        //Var3 wird an Str angehängt
   Str = strcat(Str, Extra);       //Extra wird an Str angehängt
   return Str;
}
```

Templates in großen Projekten (mit mehreren Modulen)

Deklaration und Definition eines Templates müssen sich im selben Modul befinden, sonst gibt der Compiler einen Fehler aus. Auch Spezialisierungen gehören ins selbe Modul. Sinnvoll ist es, alles in Header-Dateien zu packen und, falls gewünscht, in die anderen Module einzubinden. Dabei solltest du beachten, dass (generell) Definitionen nicht mehrfach vorgenommen werden dürfen!

Hinweis

In diesem Kapitel habe ich nur einen kleinen Teil eines riesigen Themengebietes gezeigt. Falls dich Templates übermäßig interessieren, solltest du im Internet nach verschiedenen Artikeln zu Templates suchen. Weitere Themen wären z.B. Vererbung von Templateklassen, Templates in anderen Templates ("Element-Templates"), komplexe Datentypen als generische Typen (Strukturen, Klassen) oder die Template Standard Bibliothek (STL - Standard Template Library).

Links:

http://www.volkard.de/vcppkold/templateklassen.html
http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop16_2.html
http://cpp.programmersbase.net/

Kapitel 26

In diesem Kapitel wirst du eine neue Technik des Fehlerausbesserns kennenlernen.

Ausnahmebehandlung

Eine Ausnahme ist in C++ ein unerwartetes Problem, das zur Laufzeit (also nicht zur Complilierzeit) auftritt. In diesem Fall kann es sein, dass das Programm sich selbst beendet, z.B. weil das Betriebssystem es dazu zwingt (Windows: "Programm.exe verursachte einen Fehler in XYZ.dll") oder das Programm weiterläuft und unerklärliche Fehler verursacht. Fehlerquellen können dabei sein, dass das Programm auf einen unerlaubten oder nicht existierenden Speicherbereich zugreift oder unzulässige Werte verwendet (Division durch `NULL`...). Lösungsmöglichkeiten kannst du dir vielleicht schon denken:

1.) Berichtigen des Wertes (Beispiel : `if(Variable > Wert) Variable = 1028938;`)
2.) Springen zu fehlerfreier Stelle (Beispiel: `if(Variable > Wert) goto vorher; else goto nachher;`)
3.) Auslassen des Codes (Beispiel: `if(Variable > Wert) {tue das richtige;} else {Ersatzlösung;}`)

C++ bietet uns dazu noch eine bessere Möglichkeit: Exeption Handling/Ausnahmebehandlung. Dafür stehen drei neue Schlüsselwörter zur Verfügung: **`try`**, **`throw`** und **`catch`**. (Wahrscheinlich kommst du mit den obigen drei Lösungsmöglichkeiten so gut zurecht, dass dir Ausnahmebehandlung zunächst sinnfrei erscheint. Im Abschnitt "Ausnahmebehandlung bei Funktionen" wirst du aber sehen, dass diese Funktionalität durchaus Sinn macht :).

Im sogenannten **`try`**-Block wird der Code, der auf Ausnahmen geprüft werden soll, "versucht":

```
try
{
   //Code, der versucht werden soll
}
```

Wenn z.B. der gesamte Code einer Funktion in einem einzigen **`try`**-Block steht, wird eben dieser Code auf Ausnahmen getestet. Gehen wir zunächst davon aus, dass keine Ausnahmen entstehen:

```
int* Func(int *A)
{
   int Feld[*A];
   for(int i = 0; i <= *A; i++)
   {
      Feld[i] = i;
   }
   return Feld;
}
```

Im Schleifenkopf steht geschrieben, dass die Zählervariable von `NULL` bis einschließlich `*A` zählen soll. `Feld[i]` würde demzufolge im letzten Durchgang einen Wert zugewiesen bekommen, was aber eigentlich gar nicht zulässig ist (schließlich würde in einen

unerlaubten Speicherbereich geschrieben werden). Jetzt wollen wir der Sache mal auf den Grund gehen:

```
int* Func(int *A)
{
   try
   {
      int Feld[*A];
      for(int i = 0; i <= *A; i++)
      {
         Feld[i] = i;
      }
      return Feld;
   }
}
```

Eine Ausnahme wird mittels **throw** "ausgeworfen". Der Syntax ist **return** sehr ähnlich:

```
throw Ausdruck;
```

, wobei Ausdruck für eine Konstante oder eine Variable steht. **throw** kann nur einen Parameter haben, der geworfen wird; außerdem muss **throw** immer in einem **try**-Block stehen.

Nachdem eine Ausnahme ausgelöst worden ist, wird der Code, der nach der **throw**-Anweisung im **try**-Block steht, nicht mehr ausgeführt. Nun muss diese Ausnahme irgendwie "aufgefangen" werden. Das geschieht mit **catch**:

```
catch(Typ Name)
{
   //Code, der bei einer Ausnahme ausgeführt werden soll
}
```

throw wirft eine Ausnahme immer von einem bestimmten Typ aus. **catch** vergleicht den ausgeworfenen Typen mit Typ und führt bei Übereinstimmung den Code im **catch**-Block aus. Ein **catch** hat immer nur einen Parameter, egal ob einfacher oder komplexer Datentyp. Wenn Ausnahmen möglicherweise von unterschiedlichen Typen ausgelöst werden können, kannst du **catch**-Blöcke mit entsprechenden Datentypen schreiben (das Programm sucht sich den Richtigen).

Die **catch**-Blöcke müssen immer direkt auf den **try**-Block folgen; Code zwischen **try**- und **catch**-Block ist unzulässig. Damit du im **catch**-Block mit dem Parameter machen kannst, was du willst, braucht der Parameter einen Namen; der ist im **catch**-Block-Kopf mit Name angegeben.

Hier ein komplettes Muster:

```
try
{
   bool Sonderfall, Sonderfall2;
   //Code, der normal ausgeführt wird
   if(Sonderfall) throw "Ausnahme ist eingetreten";
   //Code, der normal ausgeführt wird, im Sonderfall aber
übersprungen wird
   if(Sonderfall2) throw 6556;
   //Code, der normal ausgeführt wird, im Sonderfall oder im
zweiten Sonderfall aber nicht
}
```

```
catch(const char *Str) // catch(char *Str) würde den String von
oben nicht auffangen!!
{
   cout << "Sonderfall : " << Str;
}

catch(short Value)
{
   cout << "Fehler-Nummer " << Value << " !\nFehlerstelle blabla
!";
}
```

Beispiel throw1.

Wenn allerdings keine Ausnahme ausgelöst wird, werden sämtliche **catch**-Blöcke übersprungen und das Programm läuft dann ganz normal weiter. Der Code nach den **catch**-Blöcken wird also auf jeden Fall ausgeführt werden. Wird eine Ausnahme durch keinen der vorhandenen **catch**-Blöcke aufgefangen, so wird das Programm beendet (dazu später noch ein bisschen mehr).

Das Standard-catch

Ein weiteres Konstrukt erlaubt das Auffangen sämtlicher Datentypen, die nicht durch andere **catch**-Blöcke behandelt wurden. Dabei werden einfach Datentyp und Name im **catch**-Block-Kopf durch drei Punkte ersetzt:

```
catch(...)
{
   printf("Irgendein Fehler ist aufgetreten");
}
```

Dieser Block sollte immer erst zum Schluss kommen, da sonst unbeabsichtigt eine Routine übersprungen werden könnte.

Ausnahmen weiterleiten

Wenn in einem **try**-Block sich ein weiterer **try**-Block mit all seinen **catch**-Blöcken befindet, kann der innere **try**-Block eine ausgelöste Ausnahme zum äußeren weiterleiten. Im inneren **catch**-Block wird dann einfach ein **throw;** eingesetzt:

```
try //äußerer try-Block
{
   printf("Anfang1\n");
   try
   {
      printf("Anfang2\n");
      throw "Ausnahme innen!!";
   }
   catch (const char *str)
   {
      throw; //wird weitergeleitet
   }
}
catch (const char *str2)
{
   printf("%s\n", str2);
}
```

Beispiel throw2.

Ausnahmebehandlung bei Funktionen

So, bis hierhin kannst du alles eigentlich auch ohne Ausnahmebehandlung recht einfach hinkriegen. Bei Funktionen ist die Ausnahmebehandlung allerdings wesentlich sinnvoller zu gebrauchen. Einen Fehler, der in einer Funktion auftritt, musst du nämlich nicht gleich in der Funktion auffangen, sondern kannst ihn durch die aufrufende Umgebung behandeln.

Konkret bedeutet das, dass in einem **try**-Block eine Funktion aufgerufen werden kann, die eine **throw**-Anweisung enthält. Die Ausnahme wird dann wie gewohnt nach dem **try**-Block von einem passenden **catch** aufgenommen.

Beispiel throw3.

In diesem Beispiel kann die `f1()`-Funktion nicht mehr außerhalb eines **try**-**catch**-Konstrukts ausgeführt werden, da sonst das Programm beim Auftreten der Ausnahme beendet wird.

Vor allem in Funktionen, von denen du nur die Deklaration siehst (wie in den Standardbibliotheken), ist es äußerst gemein, wenn da eine Ausnahme auftritt. Dein Programm endet dann vollkommen unerklärlich. Das kann man umgehen, indem man in der Deklaration der Funktion die möglicherweise ausgeworfenen Typen aufzählt:

```
void func(void) throw (int, double, char);
```

Nun weiß man, dass `func()` Ausnahmen von den Typen **int**, **double** oder **char** auswerfen könnten. Der Syntax ist einfach ein **throw**, dem in runden Klammern die Datentypen (getrennt durch Kommas) folgen. Die Liste muss vollständig sein, kann aber auch zu groß sein. Auf jeden Fall empfehle ich dir, die Liste der Deklaration und Definition immer anzuhängen, ohne einen Typ zu vergessen (sonst wird eine Ausnahme des Typs **bad_exception** ausgelöst).

So können wir die `f1()`-Funktion besser definieren:

```
void f1(void) throw (const char*)
{
   int Array[10];
   for(int i = 0; i <= 10; i++)
   {
      if(i > 9) throw "f1 loest Ausnahme aus!!";
      Array[i] = i;
      printf("%d\n", i);
   }
   printf("wird nicht ausgegeben");
}
```

Ausnahmen, die der Standard definiert

Beim Compiler dabei sind 2 Bibliotheken, die uns einige "Ausnahmeklassen" und Funktionen vordefinieren. Das sind **exception** und **stdexcept** (jeweils ohne *.h-Dateiendung), für ein paar Klassen aber auch andere Header. In **exception** ist eine gleichnamige Klasse definiert:

```
class exception
{
   public:
      exception () { }
      virtual ~exception () { }
```

```
        virtual const char* what () const;
};
```

Dies ist die Basisklasse für eine ganze Hierarchie. Die `what()`-Funktion liefert ein kurze Beschreibung der aufgetretenen Ausnahme.

Im Header stdexcept findest du außerdem Definitionen für folgende Klassen:

Basisklasse	Klasse (von `exception` abgeleitet)	davon abgeleitet	ausgelöst von
`exception`	`logic_error`	`domain_error`	
		`invalid_argument`	
		`length_error`	
		`out_of_range`	
	`runtime_error`	`range_error`	
		`overflow_error`	
		`underflow_error`	
	`bad_alloc` (Header: **`new`**)		**`new`** oder **`new[]`**
	`bad_cast` (Header: **`typeinfo`**)		**`dynamic_cast<>()`**
	`bad_typeid` (Header: **`typeinfo`**)		**`typeid()`**
	`bad_exception` (Header: **`exception`**)		unerwarteter Ausnahme in Funktionen

Insgesamt gibt es also 14 schon definierte Klassen. Die Klassen `logic_error` und `runtime_error` und deren Ableitungen stehen nur so zur Verfügung und werden teils sogar von (wenigen wenigen) Standardbibliotheksfunktionen verwendet. `bad_alloc` aus dem Header **`new`** wird ausgelöst, wenn etwas beim Anlegen eines dynamischen Speichers mit **`new`** oder **`new[]`** schief läuft (sehr sehr selten). `bad_cast` findest du im 12.ten Kapitel erklärt. `bad_typeid`: siehe weiter unten. `bad_exception` wird ausgelöst, wenn in einer Funktion ein Typ geworfen wurde, der nicht in der Ausnahmen-Liste der Funktion aufgeführt wurde (Details im vorigen Abschnitt).

Gut so: alle dieser Klassen sind von `exception` abgeleitet, sodass es eigentlich ausreicht, ein **`catch`**`(exception xyz)` zu schreiben. Mit einer Ausgabe des Strings, den `xyz.what()` zurückgibt, erfährst du dann die Ursache.

Reaktionen auf Ausnahmen

Mithilfe einiger Funktionen aus dem Header exception können wir die Reaktion des Programms auf unerwartete oder unaufgefangene Ausnahmen auch selber bestimmen. Normalerweise wird, wenn eine Ausnahme nicht gefangen wird, das Programm mit `terminate()` (diese ruft `abort()` auf) beendet, bzw. `unexpected()`, wenn eine Funktion einen in ihrer Schnittstelle nicht angegebenen Typ auswirft. Letztere wirft wiederum die oben erwähnte Klasse `bad_exception` aus.

Die Funktion `set_terminate()` erlaubt es uns, eine neue Funktion für nichtaufgefangene Ausnahmen zu melden. Als einzigen Parameter verlangt sie einen Funktionszeiger auf die neue Funktion und gibt einen Zeiger auf die alte Funktion zurück (das ist `terminate()`).

Die neue Funktion kommt nicht drum rum, das Programm zu beenden. Nachdem sie einige "Aufräumarbeiten" getan hat, sollte sie z.B. `exit(-1)` aufrufen. Sonst wird das Programm automatisch beendet.

Beispiel throw4.

`set_unexpected()` macht dasselbe bei den unerwarteten Ausnahmen. Die normale Funktion ist `terminate()`.

Deine Funktionen dürfen jeweils keinen Parameter verlangen sowie keinen Rückgabetyp haben.

Workshop

Zeit für ein paar Fragen

1.) Muss einem **`try`**-Block ein **`catch`**-Block folgen?
2.) Muss ein **`try`**-Block vor einem **`catch`**-Block stehen?
3.) Was darf nicht in einem **`try`**-Block?
4.) Gibt es ein Universal-**`catch`**??
5.) Was tut die Funktion `what()`??
6.) Was bringt Ausnahmebehandlung, das andere Lösungswege nicht bringen???

Links:

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop19.html

Kapitel 24

Endlich mal ein richtig kurzes Kapitel!

Dynamische Typinformation (Run Time Type Information - RTTI)

Mit einem neuen Operator kannst du in Zukunft besser kontrollieren, was für Datentypen du verwendest. Bis jetzt konntest du nur überprüfen, was eine Variable für einen Wert hat. Mit dem Schlüsselwort **typeid** wirst du nun rauskriegen, was für ein Typ hinter der Variable steckt.

typeid geht bei einfachen sowie komplexen Datentypen. In die Klammern kommt die zu testende Variable oder sogar der Datentyp:

```
char s;
typeid(s);
typeid(char);
```

Nun, dieser Code macht noch nicht viel.

Der Operator ist eng mit einem Header verbunden: typeinfo. Darin sind die Klassen type_info sowie bad_cast und bad_typeid definiert. Zunächst ist nur type_info interessant: Jeder Ausdruck **typeid**(Variable oder Typ) ist eigentlich ein Objekt dieser Klasse. Damit stehen dir die zwei Vergleichsoperatoren != und == und die Funktion name() zur Verfügung. Die beiden Vergleichoperatoren dienen zum Vergleichen von zwei **typeid**()'s. name() gibt einen String zurück, der den Namen eines Datentyps enthält.

Beispiel typeid1.

Du kannst nun feststellen, ob ein Basisklassenzeiger auf ein Objekt der abgeleiteten Klasse zeigt:

```
class A {} pA*;
class B : public A {} B1;

//...

if(typeid(*pA) == typeid(B1)) printf("*freu*");
```

Doch diese Technik hat auch Nachteile: Für den Compiler bedeutet RTTI einen recht großen Aufwand, zudem wird das Programm ein bisschen größer. Und wahrscheinlich ist ein Programm mit RTTI ein bisschen langsamer als ohne. Auch können Eigenschaften wie **const** oder **volatile** nicht überprüft werden.

Wenn du einen NULL-Zeiger prüfen lässt, wird das Programm eine Ausnahme des Typs bad_typeid auslösen. Diese kannst du allerdings mit **catch**(exception &e) oder **catch**(bad_typeid &_bad) abfangen.

Links:

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop17_8.html

Kapitel 25

Erweiterte C++-Casts

Mit den C++-Casts können wir sehr gut kontrollieren, was passieren soll. Dafür gibt es 4 weitere Schlüsselwörter:

static_cast
const_cast
reinterpret_cast
dynamic_cast

Die haben jeweils ihr eigenes Anwendungsgebiet, da alle unterschiedlich casten. Jedoch werden sie alle gleichartig angewandt:

```
..._cast <NeuerTyp> (Ausdruck)
```

In den eckigen Klammern kommt der Datentyp, zu dem gecastet werden soll. Ausdruck steht für die Variable oder den Ausdruck, der umgewandelt werden soll.

static_cast

Der **static_cast** dient eigentlich nur dazu, den ursprünglichen C-Cast zu ersetzen. Praktisch macht

```
int x = static_cast<int>(0.302910f);
```

genau dasselbe wie

```
int x = (int)0.302910;
```

Demzufolge haben beide Casts das gleiche Anwendungsgebiet: einfache Datentypen, aber auch Zeiger auf Objekte einer Klassenhierarchie. Allerdings gilt es für C++-Programme als besseren Programmierstil, den alten Cast zu begraben und ausschließlich **static_cast** zu verwenden. Ganz egal, ob das ein bisschen mehr Schreibarbeit ist!

static_cast soll für Umwandlungen eingesetzt werden, die schon zur Übersetzungszeit sicher sind!

Beispiel static_cast1.

Einzige Einschränkung ist, dass die Attribute **const** oder **volatile** nicht weggecastet werden kann.

const_cast

Dieser befasst sich mit den Attributen **const** und **volatile**. Er kann ein Objekt so umwandeln, dass das entsprechende Attribut verschwindet. Das heißt aber jetzt nicht, dass das Objekt beliebig von dir verändert werden kann! Es dient eher dazu, in einer Funktion, der das Objekt als **const**-Objekt übergeben wurde, einer weiteren Funktion, die dieses Objekt als nicht-**const** verlangt (es aber hoffentlich nicht ändert), als Parameter zu dienen.

Beispiel const_cast1.

Hier hat die Funktion `WeitereFunktion(_A* WF_Obj)` einen nicht-**const**-Parameter, bekommt aber beim Aufruf durch `Funktion(const _A* Obj)` ein konstantes Objekt übergeben. Dies nimmt der Compiler allerdings nicht ohne **const_cast** an.

Generell ist es allerdings kein guter Stil, das **const**-sein eines Objektes zu ignorieren. Schließlich hat man sich was dabei gedacht, ein Objekt konstant zu machen.

reinterpret_cast

Bei **const**-, **static**- und **dynamic_cast** überprüft der Compiler die Gültigkeit der Umwandlung. Anders bei **reinterpret_cast**: Hier kopiert der Compiler das exakte Bitmuster eines Ausdruckes und presst es in die gewünschte Form. Die Variable wird sozusagen als anderer Typ angesehen.

Beispiel reinterpret_cast1.

Der **reinterpret_cast** ist dazu geeignet, Zeiger zu anderen Zeigern oder Zeiger zu **int**-Variablen umzuwandeln. Dabei spielt es keine Rolle, ob der erste Zeiger ein Zeiger auf ein Objekt ist und der andere auf eine einfache Variable zeigt.

Dieser Cast wird eigentlich selten angewandt. Das Resultat eines solchen Casts ist nämlich systemabhängig und außerdem kann ein seltsamer Bit-Salat herauskommen. Die Attribute **const** und **volatile** können nicht weggecastet werden (dafür gibt es schließlich **const_cast**).

dynamic_cast

dynamic_cast dient dazu, Referenzen und Zeiger auf Klassen und auf **void***
umzuformen.

dynamic_cast kann einen Zeiger oder eine Referenz auf ein Objekt `B` in einen Zeiger oder eine Referenz der Klasse `D` umformen, wobei `D` von `B` abgeleitet. Bedingungen sind:

- `D` muss von `B` abgeleitet sein (wird zur Laufzeit geprüft)
- Objekt `B` muss auf ein Objekt der Klasse `D` zeigen (oder referenzieren)

Zudem kann es einen Zeiger oder eine Referenz auf ein Objekt `D` in einen Zeiger oder eine Referenz der Klasse `B` Umwandeln.

Beispiel dynamic_cast1.

Es kann, wie alle außer **const_cast**, kein Attribut **const** oder **volatile** wegcasten.

Zusammenfassung

So, wieder ein Kapitel geschafft. Diesmal keine Fragen, aber du kannst ja trotzdem ein bisschen üben.

Nun sind auch alle C++-Schlüsselwörter erklärt (außer **__asm** für Assembler-Code, jetzt noch nicht wichtig). In den folgenden Kapiteln werde ich dir ein paar Sortier- und Verschlüsselungsverfahren erklären. Du hast jetzt schon die Grundlagen geschafft und könntest gleich bei was anderem loslegen. Wie auch immer, die folgenden Kapitel werden

dir einen Einblick in die fortgeschrittene Programmierung in Verbindung mit Mathematik geben.

Links:

http://www.math.uni-wuppertal.de/%7Eaxel/skripte/oop/oop4_2.html

Kapitel 26

Sortierverfahren

In diesem Kapitel will ich dir einige der bekanntesten Sortierverfahren erklären. Sie unterscheiden sich natürlich in Geschwindigkeit und Effizienz. Jedes kann ein beliebig großes Feld von beliebigen Daten sortieren (hier findest du aber nur Versionen für **int**'s; es wäre eine prima Übung, diese Algorithmen zu Template-Funktionen umzubauen!). Die Felder werden der Größe nach sortiert, also mit dem kleinsten Element beginnend. In den Beispielen wirst du Anweisungen zur statistischen Auswertung und zum Zeitmessen finden, damit du einen Überblick bekommst.

Die Problematik des Sortierens ist schon etwa so alt wie die moderne Computergeschichte selbst (angefangen mit den ersten mechanischen "Computern"). Früher musste man Daten von z.B. Volkszählungen sortieren. Daher hat man mathematische Verfahren entwickelt und dementsprechend die Maschinen gebaut.

Doch nun erst mal eine Vertauschfunktion:

```
void swap(int *p1, int *p2)
{
   int tmp = *p1;
   *p1 = *p2;
   *p2 = tmp;
}
```

Bubble Sort

Dieses Verfahren ist recht einfach zu verstehen und soll uns daher als Einstieg dienen. Während eines Durchlaufs wird dabei jedes Arrayelement mit seinem nachfolgenden Element verglichen und, wenn gewünscht, vertauscht.

In einem Durchlauf wird immer der größere Wert "weitergetragen". Das heißt, dass der größte Wert immer als erstes seinen Platz finden wird (steigt wie eine Luftblase (bubble) nach oben).

Hier nun die erste Bubble Sort-Funktion:

```
void bubblesort(int *Array, int Size)
{
   for(int i = 0; i < Size; i++)
   {
      for(int i2 = 0; i2 < Size-1; i2++)
      {
         if(Array[i2] > Array[i2+1]) swap(&Array[i2],
&Array[i2+1]);
      }
   }
}
```

Dabei ist immer darauf zu achten, dass nicht auf unerlaubte Speicherbereiche zugegriffen wird (deswegen `i2 < Size-1`)!

Beispiel bubble1.

Ein Beispiel für eine komplette Sortierung:

Durchlauf (äußere Schleife)	Element 1	Element 2	Element 3	Element 4	Element 5
Originalwert	**5**	**2**	**3**	**4**	**1**
1	**2**	**3**	**4**	**1**	**5**
2	**2**	**3**	**1**	**4**	**5**
3	**2**	**1**	**3**	**4**	**5**
4	**1**	**2**	**3**	**4**	**5**
5	**1**	**2**	**3**	**4**	**5**

Auffällig ist, dass zunächst mittelgroße Werte eins nach vorne sortiert werden, damit der größte Wert nach hinten gelangen kann. Weiterhin auffällig ist, dass nach dem zweiten Durchlauf auch schon die **4** an ihrem Platz ist, nach dem dritten Durchlauf die **3** an ihrem Platz usw.

Das lässt sich recht einfach in einem neuen Algorithmus ausdrücken:

```
void bubblesort(int *Array, int Size)
{
   for(int i = Size-1; i > 0; i--)
   {
      for(int i2 = 0; i2 < i; i2++)
      {
         if(Array[i2] > Array[i2+1]) swap(&Array[i2],
&Array[i2+1]);
      }
   }
}
```

Dies spart nun einige Überprüfungen und somit auch ein bisschen Zeit.

Beispiel bubble2.

Durchlauf (äußere Schleife)	Element 1	Element 2	Element 3	Element 4	Element 5
Originalwert	**5**	**2**	**3**	**4**	**1**
1	**2**	**3**	**4**	**1**	**5***
2	**2**	**3**	**1**	**4***	**5***
3	**2**	**1**	**3***	**4***	**5***
4	**1**	**2***	**3***	**4***	**5***
5	**1***	**2***	**3***	**4***	**5***

Die Werte mit Sternchen werden durch diese Verbesserung nicht mehr beachtet. Genauer gesagt überprüft **bubble2** nur die Hälfte.

Zufällig ist das Feld von oben schon nach dem vierten Durchgang vollkommen sortiert. Wenn in einem Durchgang keine Werte vertauscht wurden, soll bubblesort auch keine weiteren (sinnlosen) Durchgänge starten:

```
void bubblesort(int *Array, int Size)
{
   bool IsSwap;
   for(int i = Size-1; i > 0; i--)
   {
      IsSwap = false;
      for(int i2 = 0; i2 < i; i2++)
      {
```

```
            if(Array[i2] > Array[i2+1])
            {
               swap(&Array[i2], &Array[i2+1]);
               IsSwap = true;
            }
         }
         if(!IsSwap) break;
      }
}
```

Nur wenn in einem Durchlauf ein Austausch gemacht wurde, wird ein nächster Durchlauf gestartet.

Beispiel bubble3.

Hier werden, je nach Ordnung, nicht mehr so viele Überprüfungen angestellt.

Der Bubblesort Algorithmus ist zunächst einfach zu verstehen. Allerdings ist er mit seinen vielen Vertauschungen nicht übermäßig gut. Soll heißen, dass du ihn für kleine Felder nehmen kannst, für speicherintensive und große Felder (Umgang mit Klassen und Dateien) ist jedoch ein anderes Verfahren sicherlich besser geeignet.

Straight Selection Sort

Dieses Verfahren sucht zunächst das kleinste Element im gesamten Feld und vertauscht es dann mit dem ersten Element. Dann sucht es das Zweitkleinste und vertauscht es mit Element 2. Das wird solange durchgeführt, bis das Feld sortiert ist.

```
void sselsort(int *Array, int Size)
{
   int MinPos = 0;
   for(int i = 0; i < Size-1; i++)
   {
      MinPos = i;
      for(int i2 = i+1; i2 < Size; i2++)
      {
         if(Array[MinPos] > Array[i2]) MinPos = i2;
      }
      swap(&Array[i], &Array[MinPos]);
   }
}
```

Ein Beispiel, wie Straight Selection Sort werkelt:

Durchlauf (äußere Schleife)	Element 1	Element 2	Element 3	Element 4	Element 5
Originalwert	**5**	**4**	**2**	**3**	**1**
1	**1***	**4**	**2**	**3**	**5**
2	**1***	**2***	**4**	**3**	**5**
3	**1***	**2***	**3***	**4**	**5**
4	**1***	**2***	**3***	**4***	**5**
5	**1***	**2***	**3***	**4***	**5***

Die Werte mit Sternchen werden nicht mehr beachtet.

Beispiel select1.

Hier werden 45 mal 2 Werte verglichen und 9 Vertauschungen durchgeführt.

Dieses Verfahren ist von den Vertauschungen her wesentlich effizienter, allerdings lässt sich die Anzahl der Überprüfungen nicht so wie bei bubble3 verringern. Man könnte es noch geringfügig verbessern, indem man prüft, ob sich die zwei Positionen (ursprüngliche Position und Position des kleinsten Wertes) überhaupt unterscheiden, sonst wird der Wert mit sich selbst vertauscht:

```
void sselsort(int *Array, int Size)
{
   int MinPos = 0;
   for(int i = 0; i < Size-1; i++)
   {
      MinPos = i;
      for(int i2 = i+1; i2 < Size; i2++)
      {
         if(Array[MinPos] > Array[i2]) MinPos = i2;
      }
      if(MinPos != i) swap(&Array[i], &Array[MinPos]);
   }
}
```

Beispiel select2.

So jetzt haben wir ein schnelles und gutes Verfahren, um große Felder schnell zu sortieren.

Straight Insertion Sort

Jedes Element, das vor dem betrachtetem Element steht und größer als dieses ist, wird um eine Position nach hinten verschoben. In die freigewordene Stelle wird dann das ursprünglich betrachtete Element eingefügt. Wenn alle Elemente von vorne nach hinten einmal betrachtet, ist das Feld sortiert.

```
void sinssort(int *Array, int Size)
{
   int Pos, Val;
   for(int i = 1; i < Size; i++)
   {
      Pos = i;
      Val = Array[i];
      while((Val < Array[Pos-1]) && (Pos > 0))
      {
         Array[Pos] = Array[--Pos];
      }
      swap(&Array[Pos], &Val);
   }
}
```

Ein Beispiel, wie Straight Insertion Sort werkelt:

Durchlauf (äußere Schleife)	Element 1	Element 2	Element 3	Element 4	Element 5
Originalwert	**5**	**4**	**2**	**3**	**1**
1	**4**	**5**	**2**	**3**	**1**
2	**2**	**4**	**5**	**3**	**1**
3	**2**	**3**	**4**	**5**	**1**
4	**1**	**2**	**3**	**4**	**5**
5	**1**	**2**	**3**	**4**	**5**

Beispiel insert1.

Ein Problem, was die Bewertung dieses Verfahrens ein bisschen verhindert, ist, dass beim Weiterrücken eines Wertes durch die Schleife

```
while((Val < Array[Pos-1]) && (Pos > 0))
{
   Array[Pos] = Array[--Pos];
}
```

kein `swap()` nötig ist, aber trotzdem ein Tausch gemacht wird (eben dieses Weiterrücken). Doch auch das braucht seine Zeit. Ich würde sagen, dass ein Weiterrücken etwa `1/2` oder `1/3` eines normalen `Swaps` zählt. Doch das ist etwas schwer zu implementieren. Eine bessere Messmethode ist die Zeit, die vergeht, während z.B. 3098123 Felder sortiert werden.

Eine kleine Optimierung kann auch hier wieder vorgenommen werden: Wie bei Straight Selection Sort überprüfen, ob sich die zwei Positionen oder die Werte an den zwei Positionen unterscheiden und gegebenenfalls die Vertauschung ausführen. Wobei zweiteres eher bei komplexen Klassen mit vielen internen Vertauschungen anzuwenden wäre.

Shell Sort

Dieses Verfahren dient zur Verbesserung des Straight Insertion Sort. Das Feld zuerst in eine grobe Ordnung gebracht wird. Dann wird diese Ordnung dann immer feiner.

Im ersten Durchgang wird zum Beispiel nur jedes 10.te Element verglichen. Beim zweiten Mal wird jedes 5.te Element verglichen und beim dritten Mal nur jedes 3.te Element usw.

Der Einfachheit halber ist bei unserem ersten Shell Sort dieser Schritt halb so groß wie unser Feld. Mit jedem Durchgang soll dann nur noch der halbe Schritt getan werden (wie im Beispiel eben).

```
void shellsort(int *Array, int Size)
{
   int S = Size;
   while(S > 0)
   {
      S /= 2;

      for(int i = 0; i < S; i++)
      {
         for(int i2 = i+1; i2 < Size; i2 += S)
         {
            for(int i3 = i2-1; (i3 >= 0) && ((i3+S) < Size); i3 -=
S)
            {
               if(Array[i3] > Array[i3+S]) swap(&Array[i3],
&Array[i3+S]);
               else break;
            }
         }
      }
   }
}
```

Beispiel shell1.

Kluge Köpfe haben herausgefunden, dass `S` bei geeignetem Verlauf den Algorithmus noch ein bisschen verbessern können. So ist die Reihenfolge `1093`, `364`, `121`, `40`, `13`, `4`, `1` effizienter als z.B. `512`, `256`, `128`, `64`, `32`, `16`, `8`, `4`, `2`, `1`. Für die bessere Reihenfolge gilt:

der folgende Wert ist der Aktuelle um 1 verringert und dann durch 3 geteilt. Daraus ergibt sich:

```
S = (S-1) / 3;
```

Um nicht von `Size` auszugehen, sondern um eben die `1093`, `364`, `121`, `40`, `13`, `4` und `1` darinzuhaben, müssen wir in einer Schleife den kleinsten Wert dieser Reihe, der trotzdem noch größer als `Size` ist, herausfinden.

Beispiel shell2.

Es gibt noch bessere Reihen, allerdings sind die so krumm, dass sie sich mit Sicherheit nicht in eine mathematische Formel pressen lassen.

Quick Sort

Dieser gilt als einer der schnellsten Sortieralgorithmen. Allerdings ist er vom Code her recht schwer zu verstehen, da er rekursiv ist (Rekursion - eine Funktion ruft sich selbst auf; jeweils mit veränderten Argumenten). Das Feld wird bei einer bestimmten Stelle geteilt. Dann ruft Quick Sort sich selbst für die zwei entstandenen Teilfelder auf, bis alle Teilfelder sortiert sind.

```
void quicksort(int * Array, int l, int r)
{
   int l2 = l, r2 = r, Val;
   Val = Array[static_cast<int>((l+r)/2)];

   while(l2 <= r2)
   {
      while(Array[l2] < Val) l2++;
      while(Array[r2] > Val) r2--;

      if(l2 <= r2)
      {
         swap(&Array[l2], &Array[r2]);
         l2++;
         r2--;
      }
   }

   if(l < r2) quickSort(Array, l, r2);
   if(l2 < r) quickSort(Array, l2, r);
}
```

Beispiel quick1.

Nanu, eigentlich macht Quick Sort ja mehr Vertauschungen und Überprüfungen als der letzte Shell Sort!?! Das stimmt, solange die Felder klein sind. Aber bei großen Feldern trumpft Quick Sort enorm auf. Lass die Algorithmen mal beispielsweise Felder mit 100 Elementen sortieren. Quick Sort ist nach kurzer Zeit fertig; die anderen Verfahren brauchen wesentlich länger, auch Shell Sort.

Zusammenfassung

Dies soll dir nur als kleinen Einblick in die Informatik der Sortieralgorithmen geben, daher ist vielleicht das eine oder andere Verfahren ein bisschen zu kurz gekommen. Es gibt noch viele viele weitere Sortierverfahren, darunter Bucket Sort, Oet Sort, Tripple Sort usw. Wenn du daran interessiert bist, dann guck mal unten oder auf meine Linkseite in

diesem Tutorial. Es gibt sogar Sortierverfahren, die noch schneller sind als Quick Sort, aber das ist ja nicht wunderlich. Wichtig ist, dass du die Verfahren verstehst - wer weiß schon, wie oft man sowas braucht??

Workshop

Aufgaben

1.) Schreibe ein Programm, das dir tabellarisch zeigt, wie schnell die obigen Verfahren jeweils ein 10-, ein 100-, und ein 500-Elementiges Feld sortieren. Mit Zeitmessung und allem Drum und Dran.
2.) Erweitere dieses Programm durch Sortieralgorithmen, die nicht in diesem Tutorial vorkommen (siehe Links).
3.) Überlege dir (für ein kleines Feld) eine Reihenfolge der Schrittgrößen für Shell Sort, die das Feld schneller sortiert als `3*n+1`. Versuch es wenigstens.
4.) Forme einen oder mehrere Algorithmus in eine Template-Funktion um. Benutze für Strings die Vergleichsfunktion der Standard-Bibliothek.

Links:

http://www.volkard.de/vcppkold/bubble_sort.html
http://www.volkard.de/vcppkold/straight_selection_sort.html
http://www.volkard.de/vcppkold/straight_insertion_sort.html
http://www.volkard.de/vcppkold/shell_sort.html
http://www.sortieralgorithmen.de/

Kapitel 27

Verschlüsselungverfahren

Die Verschlüsselung von Informationen ist schon recht früh eingesetzt worden. Schon die Griechen haben so was gemacht. Außerdem sehr bekannt ist, dass Caesar seine Pläne und Botschaften des öfteren verschlüsselt weggeschickt hat. Heutzutage gibt es sehr gute Verfahren, die z.B. die Geheimdienste einsetzen. Dennoch gibt es immer wieder Lücken, die die verschlüsselten Daten verraten. Man ist immer noch auf der Suche nach dem weltrettenden unknackbaren Super-Verschlüsselungsalgorithmus...

In diesem Kapitel will ich dir ein paar Algorithmen vorstellen. Natürlich unterscheiden sich die verschiedenen Methoden voneinander in Sicherheit und Geschwindigkeit. Mir geht es jedoch nicht darum, dir den Sichersten aller Sicheren zu zeigen, sondern dir ein paar einfache aber trotzdem verlässliche Verfahren anzubieten. Die kannst du dann verwenden, um etwa Speicherstände von Spielen, Passwörter oder Dokumente geheim zu halten.

Jedem Verschlüsselungsprogramm in diesem Tutorial soll gemein sein, das es eine Datei liest, dann verschlüsselt und dann in eine andere Datei schreibt. Außerdem ist für das Ver- und Entschlüsseln ein Schlüssel nötig (haha, sonst wäre es ja nicht verschlüsselt). Den Code werde ich ausschließlich in die Beispiele packen, deshalb ist dieser Kapitel etwas kürzer.

Caesar-Chiffre

Das erste Verfahren soll das des Caesars sein. Es ist nicht allzu sicher, aber es ist zunächst einfach zu verstehen. Beim Caesar-Chiffre wird jeder Buchstabe (a-z bzw. A-Z) um eine bestimmte Zahl weiter nach vorne oder nach hinten verschoben. Zum Beispiel:

```
"Hallo, wie geht's?"
```

würde um zwei Positionen weiter verschoben bedeuten:

```
"Jcnnq, ykg igjv'u?"
```

Wenn ein 'Z' um eine Position weiter verschoben werden soll, wird es logischerweise zu einem 'A'. Alle Zeichen außer die Buchstaben bleiben unberührt, schließlich wollte Caesar damals nur seine lateinischen Wörter verschlüsseln, nicht aber seine Fragezeichen, Kommas usw.

Beispiel caesar1.

Ein sinnvoller Schlüssel liegt im Bereich 1 bis 25. 25 könntest du allerdings auch durch -1 realisieren. Wie du siehst, gibt es hier also nicht allzu viele Möglichkeiten.

Substitutions-Chiffre

Hier wird jeder Buchstabe durch einen anderen Buchstaben ersetzt. Das heißt, dass jeder Schlüssel 26 Buchstaben haben muss (haben sollte). Diese Tatsache ist wahrscheinlich ein großer Schwachpunkt, lässt sich allerdings trotzdem vermeiden. Eine weitere Schwachstelle ist, dass bei genauerer Betrachtung die Häufigkeit der einzelnen

Buchstaben auffällt. Schließlich kommt das e in der deutschen Sprache recht oft vor (etwa 17% von allen Buchstaben).

Beispiel sub1.

Schon wenn du alle Buchstaben um eine Position nach rückst und das z anstelle des ursprünglichen a setzt, wird der Text nicht mehr leserlich. Ein schlechter Substitutionscode aber ist ein normales Alphabet mit nur ein paar Vertauschungen.

Wenn der Substitutionscode nur teilweise angegeben ist, kannst du ihn mit geeignetem Code komplett machen. Im Beispiel wäre dann der Stringanfang von `Sub "XYZ"`.

XOR-Verschlüsselung

Diese basiert komplett auf dem logischen `XOR`-Operator (`Var1 ^ Var2`), wobei `Var1` der zu verschlüsselnde Wert und `Var2` der Schlüssel ist. Der gesamte Schlüssel kann ein String oder auch nur ein einzelnes Zeichen sein. Der `XOR`-Operator hat die Eigenschaft, wenn das verschlüsselte Zeichen noch mal mit dem Schlüssel verknüpft wird, dieses Zeichen zu entschlüsseln.

Beispiel xor1.

Angemerkt sei bei diesem Beispiel mit Dateilesen und -schreiben, dass der verknüpfte Wert noch mit `0xFF` oder `255 AND`-verknüpft werden muss, da sonst nicht komplett entschlüsselt werden kann. Dies ist bei einer String-Variante nicht nötig.

Workshop

Aufgaben

1.) Modifiziere jede hier vorgestellte Verschlüsselung so, dass nur bestimmte Zeichen verschlüsselt werden!
2.) Schreibe für die XOR-Verschlüsselung eine Variante mit einem längeren Schlüssel!
3.) Kombiniere einige Verfahren zu einem Gemisch.
    a.) Alle Zeichen sollen verschlüsselt werden.
    b.) Jedes Einzelverfahren soll bestimmte Zeichen verschlüsseln.

Links: keine.

Ende

Das Ende kommt zuerst

So, hier endet dieses Tutorial erst einmal. Wahrscheinlich werde ich an diesem Werk noch die eine oder andere Änderung vornehmen, denn es ist bestimmt noch nicht perfekt.

Zugegeben, es war (auch) für mich nicht immer leicht, alles nachzuvollziehen. Doch ich hoffe, dir das Programmieren mit C++ ausreichend nahe gebracht zu haben, damit du nun in fortgeschrittene Programmierung einsteigen kannst. Ich habe darauf geachtet, möglichst Vieles mit einzubinden und es ausreichend umfangreich zu beschreiben, dass ein Tutorial dabei herauskommt, aber auch ausreichend kurz, damit es dir als Überblick oder Nachschlagewerk dient. Das Ganze noch etwas mit Stil verbunden (ich find die zwei Schriftarten einfach passend), und ein recht ansehnliches Dokument ist entstanden.

Wie soll ich weitermachen??

Kommt drauf an, worauf du Lust hast. Wenn du von solchen Grundlagen der Programmierung nicht genug bekommst, solltest du unbedingt noch andere Tutorials durchlesen. Oder du gehst zu (wesentlich) schwereren Assembler-Programmierung, was ich allerdings nicht empfehle. Auf jeden Fall solltest du in Windows-Programmierung (oder Linux-Programmierung) einsteigen, wenn auch nur die einfachen Dinge. Denn dies eröffnet dir ein Fenster in die Spieleprogrammierung mit beispielsweise DirectX oder OpenGL (für Linux gibt es natürlich Alternativen dazu). Oder du entwickelst dein eigenes Betriebssystem ... (klingt verlockend, nicht wahr?)

Also hier mal ein paar Links:

Art	Link	Sprache
C++	**cprogramming.com**	English
C++	**home.unix-ag.org/martin/c++.ring.buch.html**	Deutsch
C++	**volkard.de/vcppkold/inhalt.html**	Deutsch
C++	**notizen-zu-cpp.de/**	Deutsch
C++	**http://www.schornboeck.net/ckurs/inhalt.htm**	Deutsch
C++	**http://www.informit.de/books/c++21/index.html**	Deutsch
C++	**codeproject.com/**	Englisch
C++	**http://www.math.uni-wuppertal.de/~axel...**	Deutsch
C++	**ca-osi.com/**	Englisch
Sammlung C++	**cppcentral.de**	Deutsch
Sammlung C++	**c-plusplus.de**	Deutsch
Sammlung C++	**robsite.de**	Deutsch
Sammlung C++ & Windows	**tutorialpage.de/**	Deutsch
Windows	**winprog.org/**	Englisch
Windows	**germandevnet.de/html/faq/cpp/winapi/index.htm**	Deutsch
Windows mit MFC	**mut.de/media/buecher/VCPLUS6/data/start.htm**	Deutsch
Spiele	**gamedev.net/**	Englisch
Spiele	**gamedevpage.de/**	Deutsch
Spiele	**spieleentwickler.org**	Deutsch
Spiele	**husser.de**	Deutsch

Für weitere Informationen such mal bei Google. Wenn du so was wie Kazaa hast, kannst du nach eBooks suchen (*.pdf; meist Englisch).

Noch ein paar Links:

Betreff:	Link	Sprache
Sortieralgorithmen	**Sortieralgorithmen.de**	Deutsch
	iti.fh-flensburg.de/lang/algorit...	Deutsch
	informatik.fh-hamburg.de/~o...	Deutsch
Verschlüsselungsalgorithmen	**ti.informatik.uni-tuebingen.d...**	Deutsch
	anujseth.com/crypto/	Englisch
	cryptool.de/	Deutsch

Das Programm CrypTool von cryptool.de veranschaulicht eine große Menge an Verschlüsselungsalgorithmen, sowohl die einfachen (in diesem Tutorial) als auch komplizierte wie AES und Blowfish. Dieses Programm zu installieren lohnt sich!

Noch ein Wort zum Compilern: **Optimiertes Compilern**

Danke an:

- Sebastian Porstendorfer für seinen Aufsatz "Was einen Programmierer ausmacht"!
- die Thron-Sippe
- das Internet, wo ich das eine oder andere nützliche Stückelchen Information herbekam
- alle, die gegen Umweltverschmutzung und Verschwendung sind

- das Fehler-Beseitigungs-Sonder-Einsatz-Kommando:

Erhard Henkes, Andreas Schüßler, Andrej D., Petter, Timo Reitz, Markus Schulz, Stefan

Kontakt

Zu erreichen auf meiner **Homepage** im Forum, Gästebuch oder dem eMail-Formular

ENDE