C++ in Übersichten

Material zum C++ Kurs

Maximilian Starke
Student der TU Dresden
Fakultät Informatik

25. April 2017

Vorwort

Dieses Skript **C++ in Übersichten** enthält Material zum C++ Kurs, den ich im Sommersemester 2017 halte (`ifsr.de/kurse`) Das Skript wird parallel zum Kurs erstellt und erweitert. Es besteht daher momentan noch als Entwurf.

Das Skript dient vordergründig als Nachschlagewerk für den C++ Kurs und besteht im Wesentlichen aus vier Kapiteln zu Einrichtung, Datentypen, strukturierter Programmierung und Randfeatures. Dabei wurde in erster Linie eine Einteilung nach logischen Zusammenhängen der Sprache C++ angestrebt, zweitrangig nach pädagogisch sinnvoller Reihenfolge. Das stellt mehr oder weniger eine hinreichende und zugleich notwendige Bedingung für die parallele Abarbeitung von Kapitel 2 und 3 dar, da Kenntnisse über Datentypen und Mechanismen strukturierter Programmierung an vielen Stellen wieder eine Einheit bilden und ineinander greifen.

Inhaltsverzeichnis

1 Einrichtung **4**
- 1.1 ISO C++ . . . 4
  - 1.1.1 Allgemeines . . . 4
  - 1.1.2 Versionen . . . 4
- 1.2 Dateien in einem C++ Projekt . . . 5
- 1.3 Compiler . . . 6
- 1.4 IDEs . . . 7
  - 1.4.1 JA oder NEIN . . . 7
  - 1.4.2 IDEs im Überblick . . . 7
- 1.5 Referenzen . . . 7
- 1.6 The Hello World . . . 12
  - 1.6.1 Das erste kleine Programm . . . 12
  - 1.6.2 Ein paar Werkzeuge . . . 12
  - 1.6.3 Programmierstil . . . 13

2 Datentypen in C++ **14**
- 2.1 Identifier . . . 14
- 2.2 primitive Datentypen . . . 14
  - 2.2.1 Die Datentypen . . . 15
  - 2.2.2 Literale . . . 16
  - 2.2.3 Initialisierung . . . 19
  - 2.2.4 Deklaration und Definition (vereinfacht) . . . 19
  - 2.2.5 Einige Operatoren auf primitiven Datentypen . . . 19
- 2.3 Konvertierungen von Typen . . . 20
- 2.4 Casts . . . 20
- 2.5 Zusammengesetzte und konstruierte Datentypen . . . 20
  - 2.5.1 Pointer . . . 20
  - 2.5.2 Referenzen . . . 21
  - 2.5.3 Arrays (C-like) . . . 22
  - 2.5.4 Strings . . . 24
  - 2.5.5 Records . . . 24
  - 2.5.6 Containerklassen . . . 24
- 2.6 Klassen . . . 24
  - 2.6.1 Konstruktoren . . . 24
  - 2.6.2 Vererbung . . . 24
  - 2.6.3 Polymorphie . . . 24
- 2.7 typedef und using . . . 24
- 2.8 const und constexpr . . . 24
- 2.9 auto . . . 24

3 Strukturierte Programmierung **25**
- 3.1 Kontrollstrukturen . . . 25
  - 3.1.1 Sequenzen . . . 25
  - 3.1.2 Sprünge . . . 26
  - 3.1.3 if - Verzweigung . . . 27
  - 3.1.4 switch - Mehrfachverzweigung . . . 27
  - 3.1.5 while . . . 27

3.1.6 do while . . . 28
3.1.7 for (klassisch) . . . 28
3.1.8 for (foreach) . . . 28
3.2 Funktionen . . . 28
3.3 Operatoren . . . 28
3.4 Modularisierung . . . 28

4 Zusätzliche Features **29**
4.1 Templates . . . 29
4.2 Exceptions . . . 29
4.3 Multithreading . . . 29

5 Ein Blick in Bibliotheken **30**
5.1 Wertebereiche mit ¡limit¿ . . . 30

Kapitel 1

Einrichtung

1.1 ISO C++

1.1.1 Allgemeines

- ab 1979 von Bjarne Stroustrup bei AT&T entwickelt als Erweiterung der Programmiersprache C
- später von ISO genormt

- effizient und schnell - Schnelligkeit eines der wichtigsten Designprinzipien von C++
- hohes Abstraktionsniveau u.a. durch Unterstützung von OOP
- ISO Standard beschreibt auch eine Standardbibliothek
- C++ ist **kein** echtes *(und auch kein unechtes)* Superset von C (siehe stackoverflow.com, ...)
- C++ ist (wie C) **case sensitive**

- Paradigmen:
  - **generisch** (durch Benutzung von Templates, automatische Erstellung multipler Funktionen für verschiedene Datentypen)
  - **imperativ** (Programm als Folge von Anweisungen, Gegenteil von deklarativ siehe Haskell und Logikprogrammierung)
  - **objektorientiert** (Klassen, Objekte, Vererbung, Polymorphie, Idee: Anlehnung an Realität)
  - **prozedural** (Begriff mit verschiedenen Bedeutungsauffassungen, Unterteilung des Programms in logische Teilstücke (Prozeduren), die bestimmte Aufgaben / Funktionen übernehmen)
  - **strukturiert** (prozedural und Teilung in Sequenz, Verzweigung, Wiederholung, ...)
  - **funktional** (ab C++11, Definitionskleinkram, siehe Wikipedia, Programm als verschachtelter [...] Funktionsaufruf organisierbar, eher typisch für Haskell o.ä.)

1.1.2 Versionen

- C++98
- C++03
- C++11

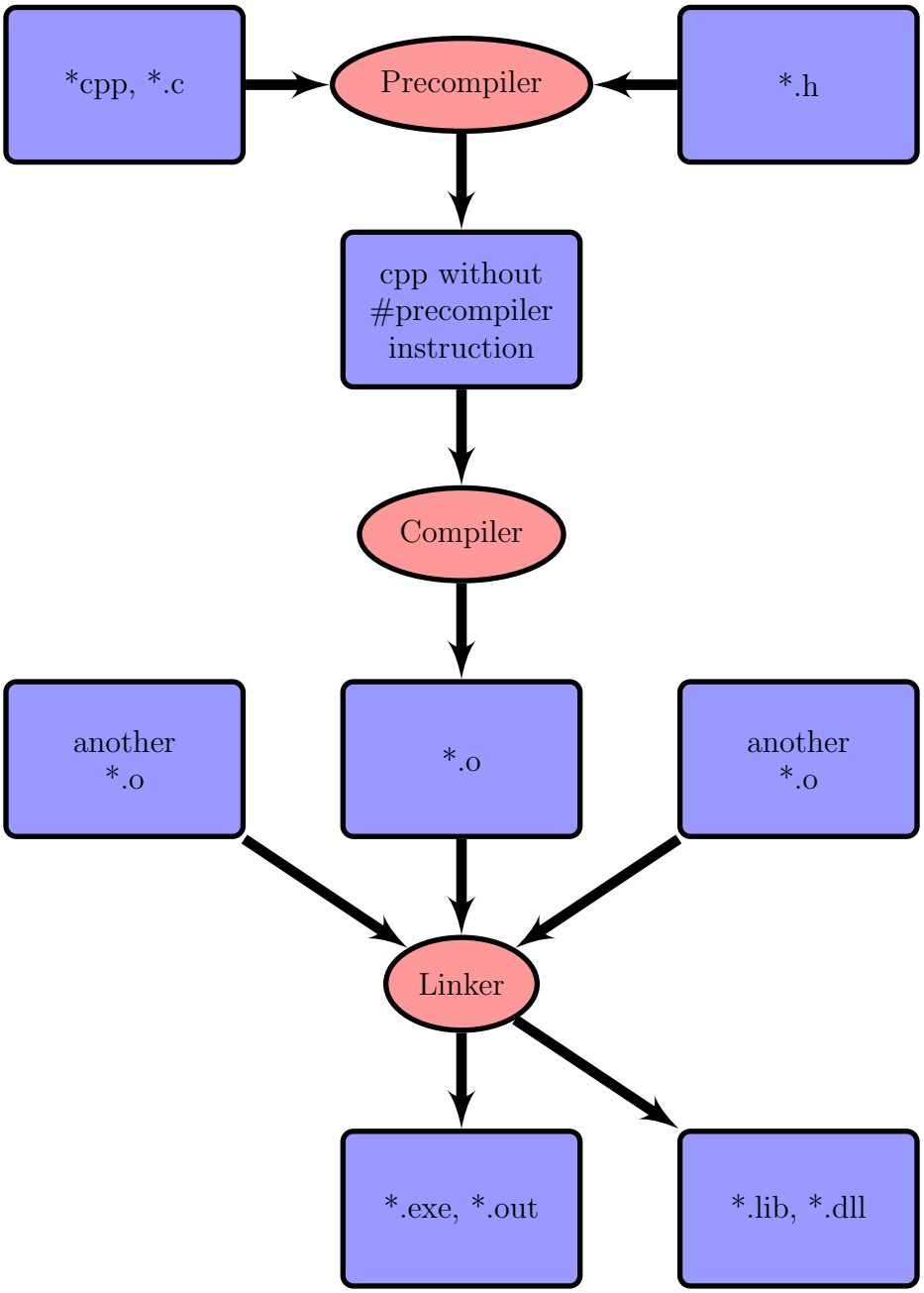
  wesentliche Neuerungen. Einführung von constexpr, Elementinitialisierer, ... Neue Bedeutung des Schlüsselworts auto # Referenzen ergänzen
- C++14

  aufweichung der constexpr Bedingungen.
- C++17

  soll 2017 vollendet werden.

1.2 Dateien in einem C++ Projekt

Dateiendung	Bezeichnung	Inhalt
(*.cpp) (*.cc)	Quelldatei	Funktionsimplementation, Klassenimplementation, Berechnungen bzw. eigentliche Arbeit erledigen
(*.h)	Headerdatei	Funktionsdeklaration, Klassendefinition, Bezeichner öffentlich bekannt machen
(*.o)	Objektdatei	Objektcode (Maschinencode) einer Übersetzungseinheit
(*.exe) (*.out)	ausführbare Datei	fertiges Programm
(*.sln) (*.pro) (*.vcxproj)	"Projektdatei"	IDE Einstellungen (oder ähnliches) IDE-spezifische Namen und Verwendung
(*.res)	Ressourcendatei	multimediale Inhalte



1.3 Compiler

Compiler	Plattform
GCC/g++	Windows, Linux, Mac, Unix-like
Clang	Unix-like, Mac, Windows, Linux
Intel-C++	Linux, Windows, Mac
VC++	Windows

Das nun folgende Listing zeigt, wie ein C++ Quellcode, der als Datei vorliegt, „per Hand“ mit Kommandozeile unter Nutzung des Compilers (hier g++) übersetzt werden kann. Beim Aufruf des Compilers wurden hier noch einige Flags gesetzt, nämlich `-Wall`, um **sinnvolle Warnungen** ausgeben zu lassen, und `-pedantic`, um **vom C++ Standard geforderte Warnungen** erscheinen zu lassen. Außerdem wurde der C++ Standard (Version) gesetzt.

Nutzung von g++ mittels Powershell

```
PS A:\> cd .\example\
PS A:\example> ls


Verzeichnis: A:\example


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       02.04.2017     08:38             87 hello_world.cpp


PS A:\example> type .\hello_world.cpp
#include <iostream>

int main(){
        std::cout << "Hello World";
        return 0;
}

PS A:\example> g++ -o programm hello_world.cpp -Wall -pedantic -Wextra -std=c++14
PS A:\example> ls


Verzeichnis: A:\example


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       02.04.2017     08:38             87 hello_world.cpp
-a----       02.04.2017     09:12          48650 programm.exe


PS A:\example> .\programm.exe
Hello World
PS A:\example>
```

Eine kleine Anmerkung zu Bezeichnungen, die mit Compilern zu tun haben, möchte ich an dieser Stelle noch loswerden, da hier immer eine kleine Verwechslungsgefahr besteht. Die **GCC** *(GNU Compiler Collection)* ist eine Compilersammlung mit Compilern zu C, C++ und einigen weiteren. Dagegen ist der **gcc** (klein geschrieben) der C-Compiler der Sammlung und **g++** der C++ Compiler.

Um auf Ihrem Betriebssystem einen C++ Compiler zum Laufen zu bringen, haben Sie meist verschiedenste Möglichkeiten.

Um unter **Linux** GCC zu nutzen, müssen lediglich die entsprechenden Pakete installiert werden, meist ist die GCC sogar schon vorinstalliert.

Unter **Windows** kann man den von Microsoft bereitgestellten Visual C++ Compiler verwenden, i.d.R. in Verbindung mit einer Installation von Visual Studio (eine IDE für u.a. C++). Die zu installierenden Kompo-

nenten bei Visual Studio kann man selbst beim Installationsprozess auswählen, i.A. ist der Speicherverbrauch jedoch relativ groß. Wer auf eine speicherschonende Variante zurückgreifen will oder muss, dem empfehle ich MinGW - eine Portierung der GCC aus dem Hause Linux für Windows. Planen Sie früher oder später Qt-Creator als eine C++ IDE zu installieren, dann können Sie sich einen extra Installation von MinGW im Vorhinein sparen, da QT-Creator MinGW bereits mitinstalliert. Sofern mit der Kommandozeile direkt auf g++ zugegriffen werden soll, ist es unter Windows i.d.R. erforderlich den Pfad der MinGW-Binarys der Systemvariablen `PATH` hinzuzufügen.

1.4 IDEs

1.4.1 JA oder NEIN

ohne IDE	**mit IDE**
Compiler, Linker über Shell bedienen Texteditor evtl. make + makefile Dokumentationen	Projekteinstellungen & Buttons in IDE integriert automatisch generiertes makefile geordneter Menübaum
Einarbeitungszeit (??) für kleine und mittelgroße Projekte	Einarbeitungszeit (??) kleine, mittlere und große Projekte

1.4.2 IDEs im Überblick

IDE	**Plattform**	**Anmerkungen**
Eclipse, Netbeans	Java (JVM)	in und für Java geschrieben, unterstützt auch C++
Qt SDK	WIN, Linux, Mac	bringt umfangreiches Qt-Framework mit für GUIs u.v.m.
Code::Blocks	WIN, Linux, Mac	
Visual Studio	Windows	kostenfreie Version für den Hausgebrauch: VS Community 2017, sehr umfangreich (Refactoring Tools, Debugger, Laufzeitanalyse, Frameworks wie MFC, ATL, WTL) und damit auch speicherintensiv, zu installierende Features wählbar, benutzt eigenen MS VC++ Compiler
Orwell DEV-C++	Windows	
Geany	Linux, WIN	schlichter Texteditor mit Syntaxhighlighting und diversen Buttons für Compilerausführung, Logausgabe
KDevelop	Linux, WIN	#
Anjuta	Linux	#
XCode	MacOS	„hauseigene“ IDE von Apple

Auf den Seiten 8 bis 11 finden sich Screenshots einiger IDEs.

1.5 Referenzen

- Buch:
  - Wolf, Jürgen: C++ - Das umfassende Handbuch. Rheinwerk Computing
- Websites:
  - `http://en.cppreference.com/w/`
  - `ttp://www.cplusplus.com/reference/`

Es gibt auch offline Versionen der Referenzen.

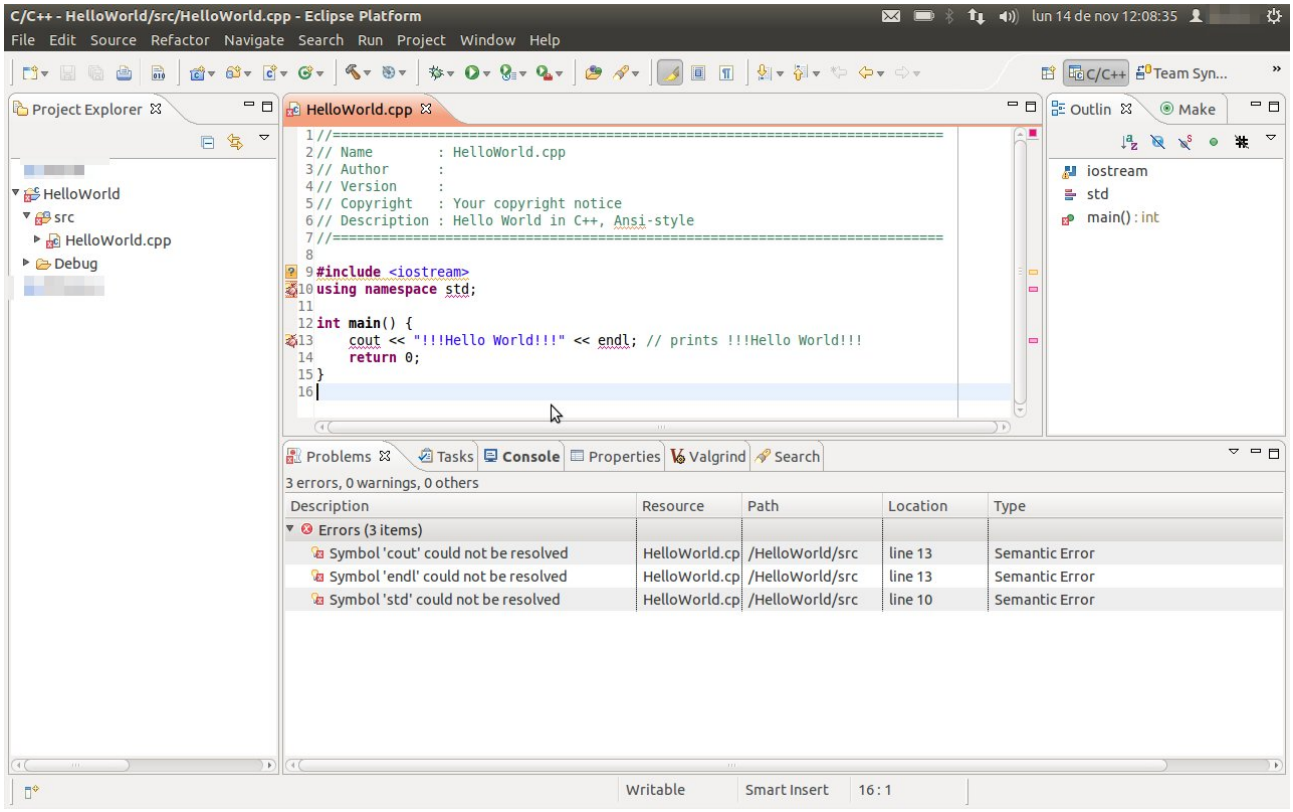


Abbildung 1.1: Eclipse mit einem C++ Projekt
https://www.eclipse.org/forums/index.php/fa/6135/0/

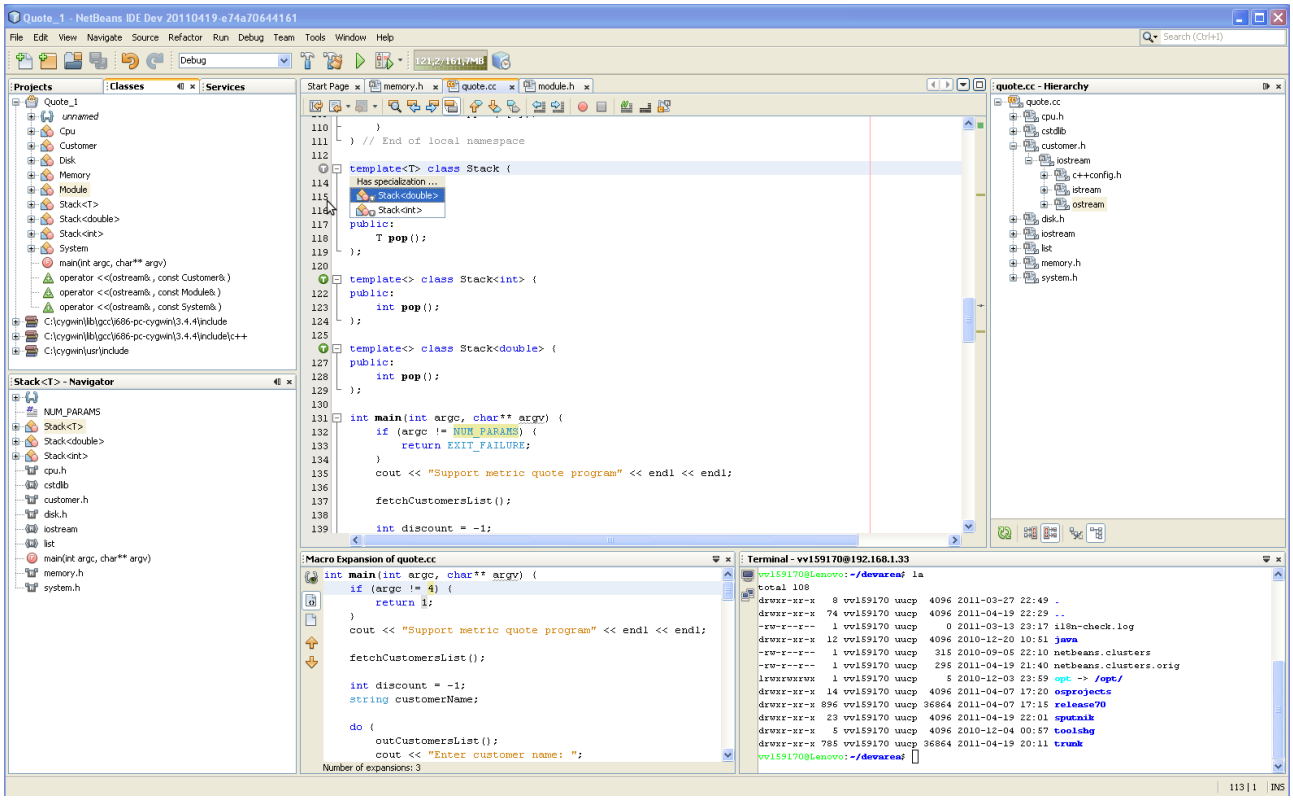
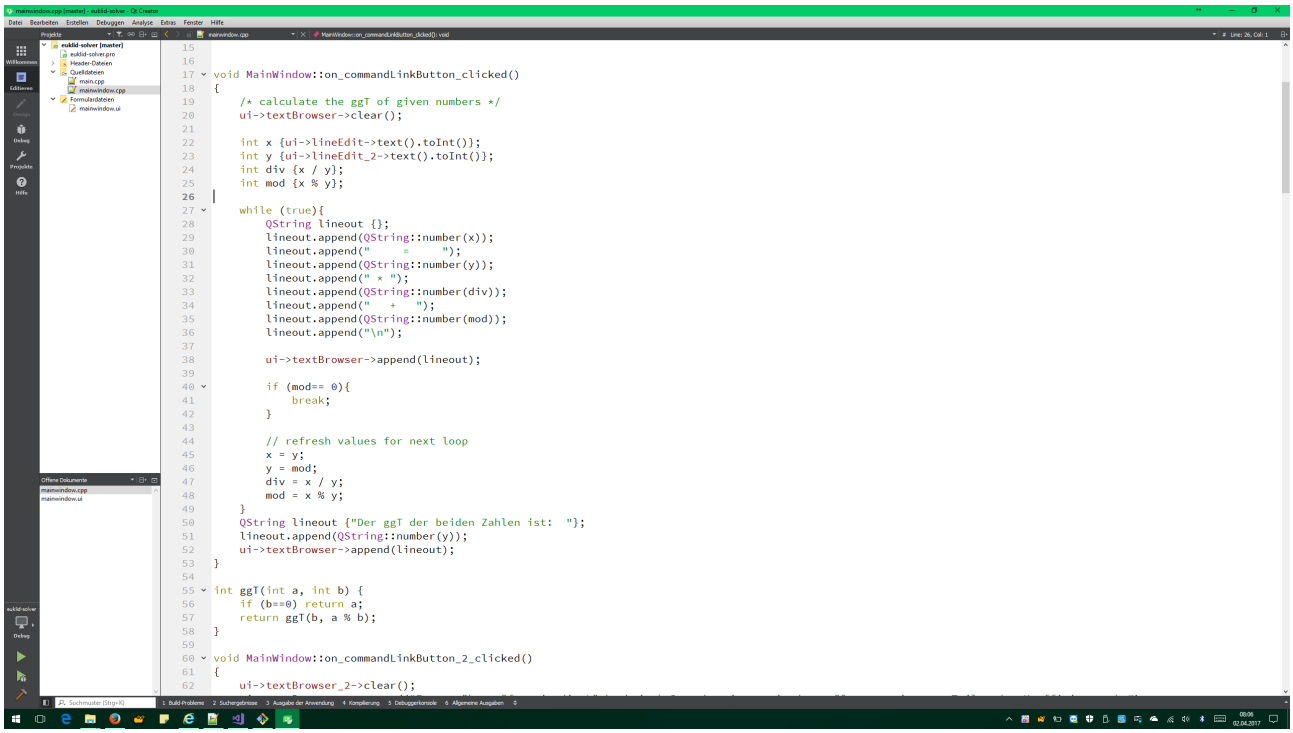


Abbildung 1.2: NetBeans und die Verwendung von C++
https://netbeans.org/images_www/v7/screenshots/cnd.png



```
void MainWindow::on_commandLinkButton_clicked()
{
    /* calculate the ggT of given numbers */
    ui->textBrowser->clear();

    int x {ui->lineEdit->text().toInt()};
    int y {ui->lineEdit_2->text().toInt()};
    int div {x / y};
    int mod {x % y};

    while (true){
        QString lineout {};
        lineout.append(QString::number(x));
        lineout.append("      =      ");
        lineout.append(QString::number(y));
        lineout.append(" * ");
        lineout.append(QString::number(div));
        lineout.append("     +     ");
        lineout.append(QString::number(mod));
        lineout.append("\n");

        ui->textBrowser->append(lineout);

        if (mod== 0){
            break;
        }

        // refresh values for next loop
        x = y;
        y = mod;
        div = x / y;
        mod = x % y;
    }
    QString lineout {"Der ggT der beiden Zahlen ist:  "};
    lineout.append(QString::number(y));
    ui->textBrowser->append(lineout);
}

int ggT(int a, int b) {
    if (b==0) return a;
    return ggT(b, a % b);
}

void MainWindow::on_commandLinkButton_2_clicked()
{
    ui->textBrowser_2->clear();
```

Abbildung 1.3: C++ Code in der QT Creator IDE

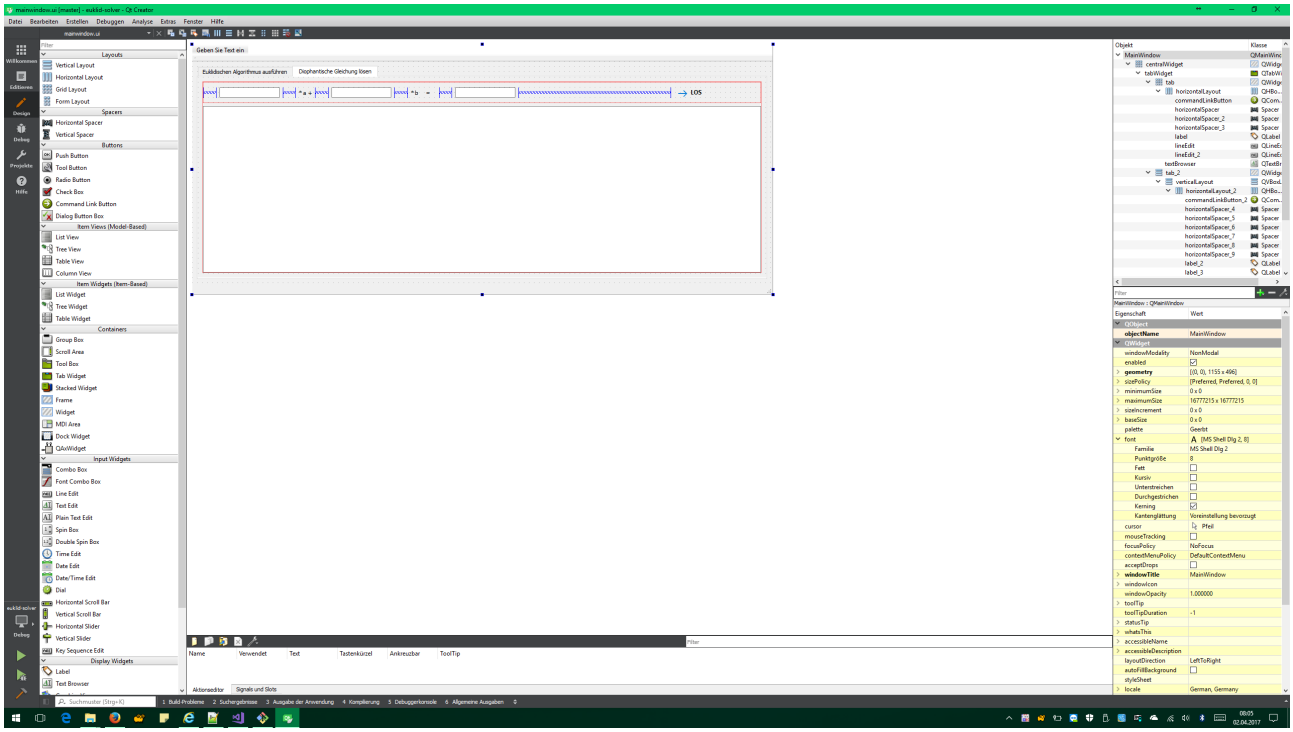


Abbildung 1.4: Fensterdesign mit QT Creator

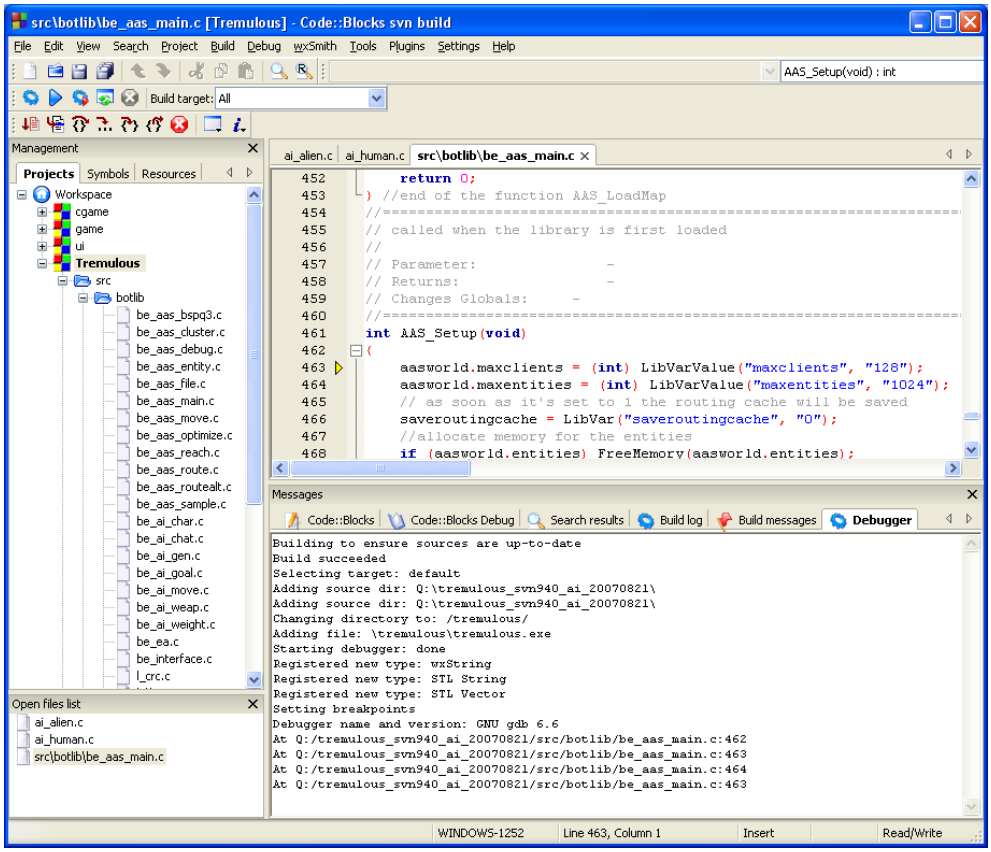


Abbildung 1.5: Code Blocks
`http://www.aftermoon.net/img/20070905_codeblocks_tremulous.png`

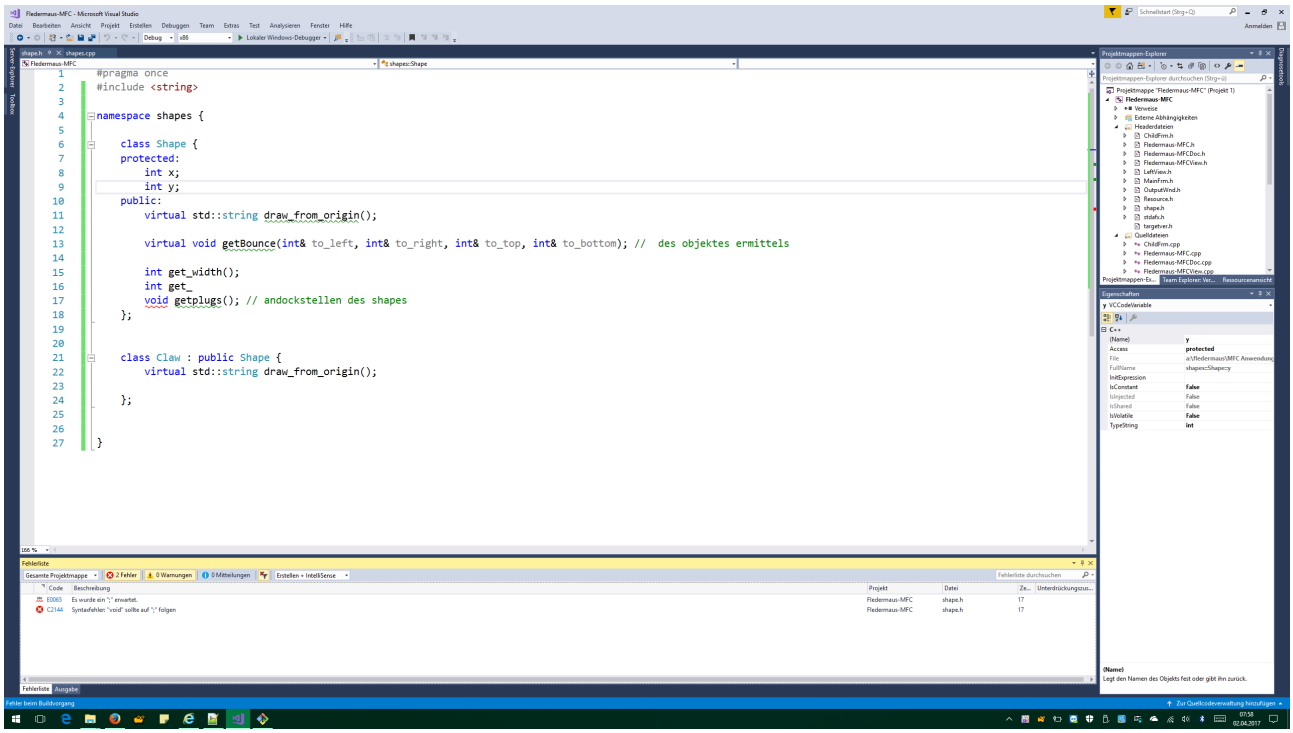


Abbildung 1.6: Visual Studio Community

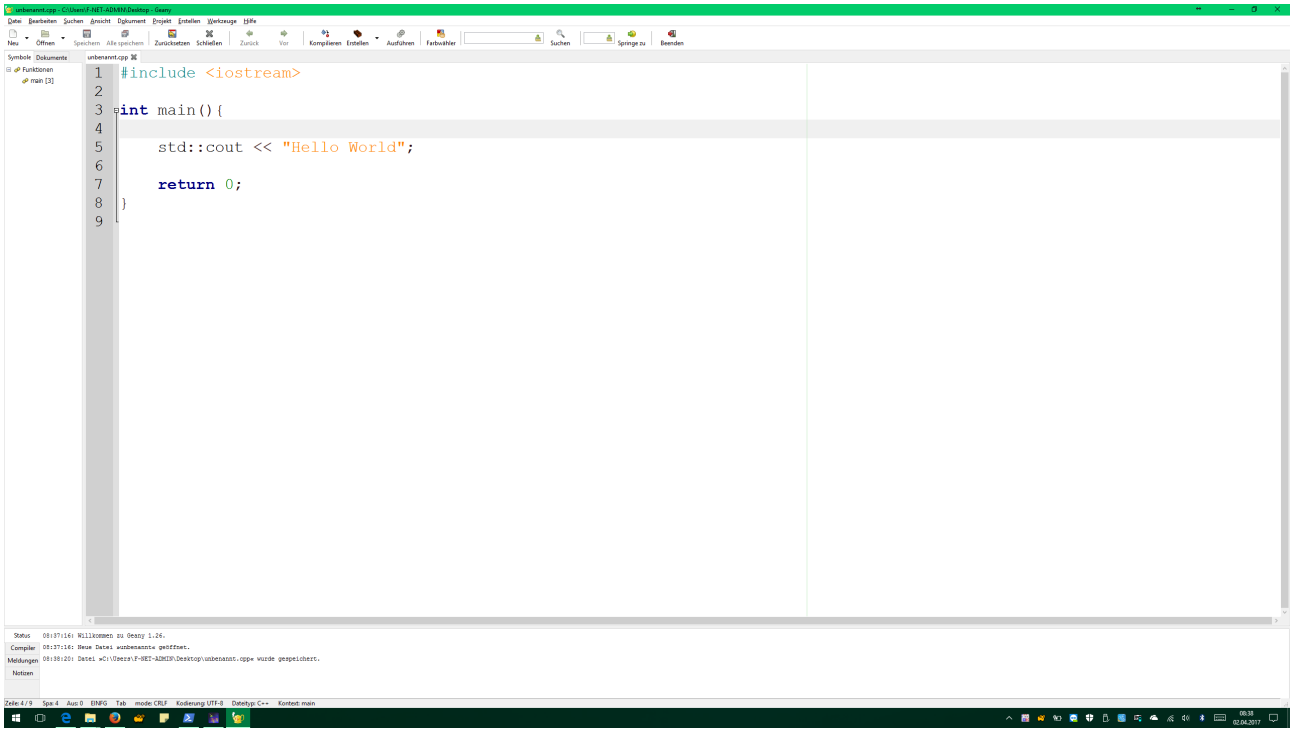


Abbildung 1.7: Geany

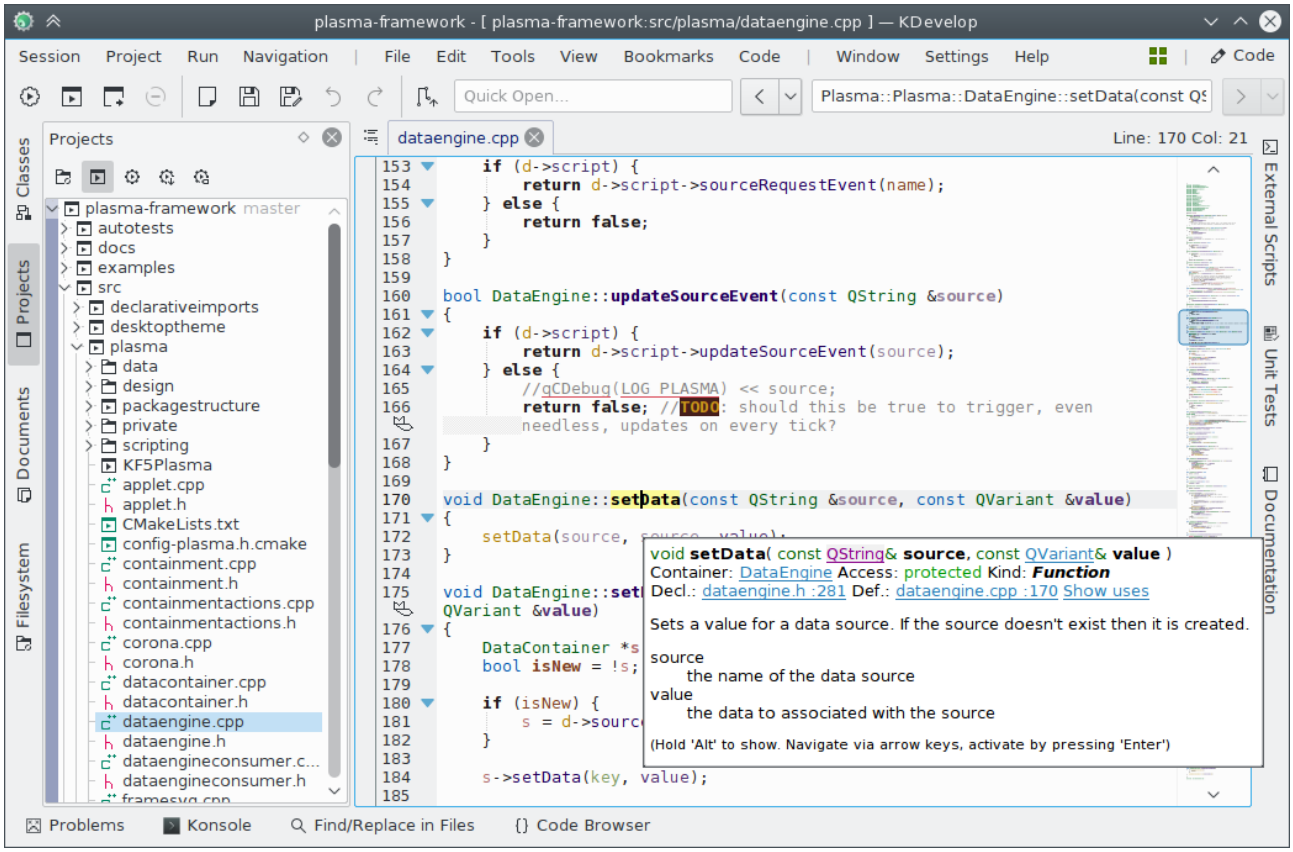


Abbildung 1.8: KDevelop
https://www.kdevelop.org/sites/www.kdevelop.org/files/inline-images/kdevelop5-breeze_2.png

1.6 The Hello World

1.6.1 Das erste kleine Programm

Unser erstes C++ Programm

```
#include <iostream>
// "Einbinden" d.h. 1-zu-1-Einfuegen des Headers iostream.h

int main(int argc, char* argv[])
// main-Funktion: Einstiegspunkt der Anwendung
// count: Anzahl der uebergebenen Parameter
// arg: Pointer auf ein Array von Pointern auf C-Style-Strings (die Parameter)
// Parameter der main-Funktion duerfen in der Signatur auch weggelassen werden.

// Parameter der main-Funktion
{   // Beginn vom Anweisungsblock der main-Funktion

        std::cout << "Hello World" << std::endl;
        // Ausgabe von "Hello World" und Zeilenumbruch
        // genauer:
        /*
        * implizite Klammerung:
        * ((std::cout) << "Hello World") << (std::endl);
        * std              ... ein Namensraum
        * ::               ... scope-Operator (Bereichsoperator)
        * cout:            ... gepufferter Standardausgabestream
        * <<               ... Ausgabeoperator (auch bitshift-Operator)
        * "Hello World"    ... C-Style-String Literal
        * endl             ... Objekt aus dem std Namensraum, das einen Zeilenumbruch ('\n')
    erzeugt.
        * ;                ... Abschluss einer einzelnen Anweisung
        */

        for(int i = 0; i < argc; ++i ){
                std::cout << i << ". Parameter:  " << argv[i] << '\n';
        } // Beipiel fuer die Ausgabe der Komandozeilenargumente
        // argv[0] ist der Name der executable Datei

        return 0; // Rueckgabewert 0 "erfolgreich (ohne Fehler) beendet"
}
```

Im Falle der `main`-Funktion ist es auch möglich das **return statement** (`return 0;`) wegzulassen. Dann wird implizit 0 als Funktionswert zurückgegeben. Die Funktionssignatur der `main`-Funktion darf auch in `int main(int argc, char** argv)` geändert werden. Der erste Arrayeintrag von `argv` enthält übrigens immer einen Zeiger auf den Namen (ohne Dateiendung), unter dem das Programm abgespeichert wurde. Damit ist `argc` stets mindestens 1.

1.6.2 Ein paar Werkzeuge

Bevor wir in Kapitel 2 einsteigen und das gesamte (naja *fast*) C++ von Grund auf kennenlernen wollen, sollten Sie noch einige nützliche Werkzeuge kennen, damit Sie neu gelernte Dinge auch ohne große Probleme ausprobieren können.

... und ein paar Hilfsmittel ...

```
#include <iostream>


#define debug // Benutzung bedingter Compilierung zum Debugging

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

        int zahl = 0;
        std::cout << "Wie alt bist du?\n"; // eine simple Ausgabe
        std::cin >> zahl; // eine simple Eingabe
        std::cout << "Okay!\n\n";

#ifndef debug
        //folgende Zeile compiliert nicht:
        std::cout << << "In 7 Jahren bist du " << 7 + zahl << " Jahre alt." << '\n';
#endif //debug

        std::cout << "Tsch" << static_cast<char>(0x81) << "ss\n";
        //https://de.wikipedia.org/wiki/Codepage_850

        std::cin.sync();
        std::cin.get(); // warted auf Enter zum fortfahren.

        /*
        Das ist
        ein mehrzeiliger
        Kommentar
        */

        // Das ist ein einzeiliger Kommentar.
}
```

Objekt	Funktionalität
`cin`	Standardeingabe, standardmäßig Eingabe von Tastatur
`cout`	(gepufferte) Standardausgabe
`cerr`	ungepufferte Standardfehlerausgabe
`clog`	gepufferte Standardfehlerausgabe
Achtung: Diese Streamobjekte liegen alle im Namensraum `std` und werden nach einem `#include <iostream>` erst verfügbar	

1.6.3 Programmierstil

Bevor es richtig losgeht, möchte ich noch ein paar Worte über den Programmierstil loswerden. Im Grunde genommen dürfen Sie Ihren C++ Code schreiben, wie sie wollen, solange Sie die Spezifikationen von c++ einhalten. Es gibt auch nicht *den einen* Programmierstil, der sich durchgesetzt hat. Sie schreiben aber einen viel leserlicheren, einfacher wartbaren und für das Auge schöneren Code, wenn Sie beim programmieren **konsistent bleiben**, was einige Aspekte betrifft:

Einrückungen	tabs or spaces
Anweisungen pro Zeile	eine, ...
Bezeichner	snake_case, camelCase, PascalCase kurz, prägnant, aussagekräftig Deutsch, Englisch, ... Isländisch

Einige IDEs können Sie sogar mehr oder weniger dabei unterstützen, in dem Sie sich um die **Quelltextformatierung** kümmern. Dies ist gerade bei Projekten mit vielen Entwicklern hilfreich, da so ziemlich effizient für einheitliches Quelltextlayout gesorgt werden kann.

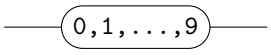
Kapitel 2

Datentypen in C++

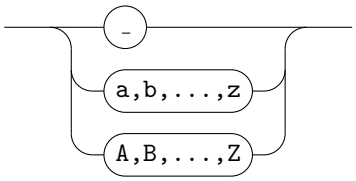
Früher oder später müssen Sie in Ihrem Programm Daten speichern, sei es während der Laufzeit im Arbeitsspeicher (RAM) oder darüber hinaus persistent in Dateien, die Sie in Dateisystemen auf zum Beispiel Festplatten aufbewahren können. Dabei steht in der Regel als erstes die **Wahl des Datentypes** im Vordergrund, denn die Wahl des Datentyps bestimmt maßgeblich die **Möglichkeiten der Verwendung der Daten**. So gibt Ihnen der Datentyp grundsätzlich vor welche Funktionen und insbesondere Operatoren Sie darauf anwenden können, beziehungsweise *was* diese bewirken.

2.1 Identifier

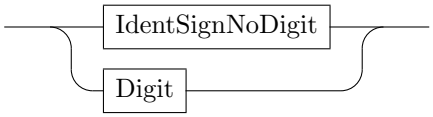
Digit



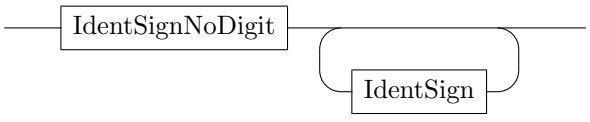
IdentSignNoDigit



IdentSign



Identifier



gültige Identifier	ungültige Identifier
`_9zig`	`9zig`
`gruen`	`grün`
`LaTeX`	`pro%zent`
`dein_alter_in_sekunden`	`Ge§etzbuch`

Anmerkungen

Beachten Sie dass für die **interne Implementierung** von C++ Bezeichner verwendet werden die mit zwei Unterstrichen (`__`) oder einem Unterstrich gefolgt von einem Großbuchstaben (z.B. `_A`) beginnen. Es wird daher ausdrücklich empfohlen, auf solche Identifier zu verzichten.

Laut Standard ist auch das **$-Zeichen** in Bezeichnern erlaubt. Auch hier wird ein Verzicht auf dieselben empfohlen, da es Compiler gab und vielleicht noch gibt, die dies nicht unterstützen.

Dagegen ist es jedoch Möglich Umlaute und viele andere **UTF-Zeichen** in Bezeichnern zu nutzen. Nicht erlaubt sind Identifier wie `schön` oder `größer`. Zeichen dürfen aber UTF-codiert in der Form `\uXXXX` als **UTF-16**-Zeichen oder in der Form `\UXXXXXXXX` als **UTF-32**-Zeichen verwendet werden. So darf statt `schön` beispielsweise `sch\u00f6n` als Identifier genutzt werden.

2.2 primitive Datentypen

Zu aller erst ist es wichtig, dass Sie mit den **eingebauten Datentypen**, auch genannt **primitive Datentypen** vertraut sind. Aus diesen setzen sich dann alle höheren Datentypen wie zum Beispiel Klassen zusammen. Auch sämtliche (oftmals relativ komplexe) Klassen aus der C++ Standardbibliothek, welche Sie zunehmend immer häufiger nutzen werden, bauen im Grunde auf nichts anderem auf.

2.2.1 Die Datentypen

Kategorien

Kategorie	Typen	Werte
integrale Typen	`short int, int, long int, long long int`	Ganzzahlen
integrale char Typen	`char, wchar_t, char16_t, char32_t`	Zeichen (enstpricht Ganzzahlen)
floating point Typen	`float, double, long double`	Gleit- bzw. Fließkommazahlen
boolsche Typen	`bool`	Wahrheitswerte

Größe

	Typ	Synonym	Größe						
			Datenmodelle bzw. Programmiermodelle						
			IP16	LP32	ILP32	LLP64	LP64	ILP64	SILP64
	`short int`	`short`	16	16	16	16	16	16	64
	`int`		16	16	32	32	32	64	64
	`long int`	`long`	32	32	32	32	64	64	64
C++11	`long long int`	`long long`	64	64	64	64	64	64	64
	`char`		≥ 8, (meist 8)						
	`wchar_t`		implementierungsabhängig: (16 oder 32)						
	`char16_t`		≥ 16						
	`char32_t`		≥ 32						
	`float`		implementierungsabhängig: $\geq 4Byte$						
	`double`		implementierungsabhängig: $\geq 8Byte$						
	`long double`		implementierungsabhängig: $\geq 10Byte$, teils $16Byte$						
	`bool`		implementierungsabhängig: $\geq 1Byte$						

Leider sind die exakten Größen der Basisdatentypen fast immer **implemtentierungsabhängig** und nicht zuverlässig voraussagbar. Es gibt verschiedene Datenmodelle für die Breite integraler Typen, wobei **I** für `Integer`, **L** für `long` und **P** für `Pointer` steht. Für **Windows 64** ist **LLP64** typisch, die meisten **unixoiden Systeme** nutzen **LP64**. Um zur Compilezeit eine Prüfung der Größe eines Datentyps durchzuführen, bieten sich der `sizeof()`-Operator und `static_assert()` an. Sie können sich jedoch allenfalls auf folgende Relationen verlassen:

- `sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
- `sizeof(float) <= sizeof(double) <= sizeof(long double)`

Wenn exakte Breiten (oder eine Mindestbreite) bestimmter Typen für Sie unerlässlich sind, können Sie mit `#include <cstdint>` eine Bibliothek importieren, die Typen fester Breite (und einiges mehr) zur Verfügung stellt:

Breite	signed	unsigned
8 bit	`int8_t`	`uint8_t`
16 bit	`int16_t`	`uint16_t`
32 bit	`int32_t`	`uint32_t`
64 bit	`int64_t`	`uint64_t`

signed and unsigned

Die Schlüsselworte `signed` sowie `unsigned` sind nur für integrale Typen von Bedeutung. Integrale Typen (ausgenommen `char`-Typen) sind standardmäßig **signed** und können daher sowohl **negative als auch positive Werte** annehmen. In diesem Fall erfolgt die Codierung mit dem 2er-Komplement. Man darf das Schlüsselwort `signed` optional auch explizit davorschreiben. Setzt man andererseits **unsigned** vor einen integralen Typ, so kann eine Variable dieses Typs nur **nichtnegative Werte** speichern, beziehungsweise ihre Werte (Bitmuster) werden als solche interpretiert.

signed	unsigned
vorzeichenbehaftete Ganzzahlen	vorzeichenlose Ganzzahlen

Eine Ausnahme bilden die **char-Typen**: Hier ist es implementierungsabhängig, ob beispielsweise eine `char` standardmäßig als `signed char` oder als `usigned char` implemetiert ist. Deswegen muss (*sollte*) in jedem Fall `signed` bzw. `unsigned` vor einen solchen Typ gesetzt werden, sofern ein solcher verglangt wird. Auch wird deswegen empfohlen, die `char`-Typen nicht für Zahlenarithmetik sondern nur für die Darstellung von Zeichen zu nutzen.

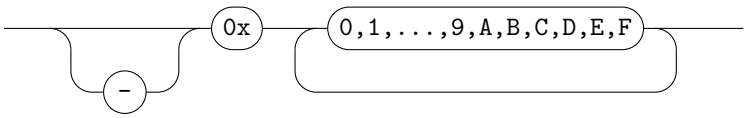
Wertebereiche

Typen	Wertebereich
`signed` Integer der Breite n bit	$-2^{n-1}, \ldots, -1, 0, 1, \ldots, 2^{n-1} - 1$
`unsigned` Integer der Breite n bit	$0, 1, \ldots, 2^n - 1$
`bool`	true (1), false (0)
`float`	
`double`	man benutze `<cfloat>`
`long double`	siehe spätere Kapitel

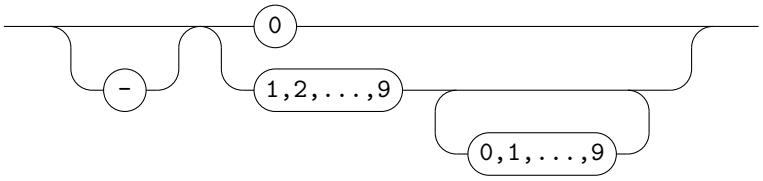
2.2.2 Literale

Integrale Typen

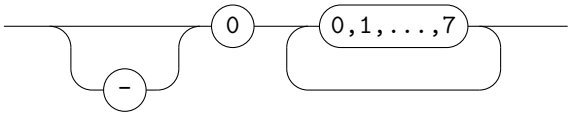
LiteralIntHex



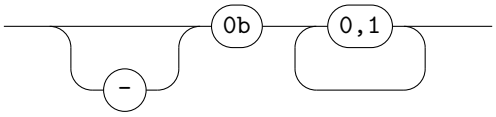
LiteralIntDec



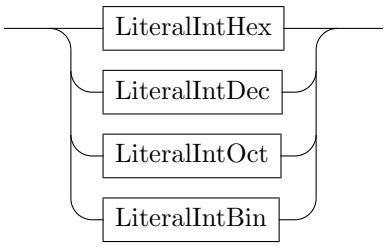
LiteralIntOct



LiteralIntBin



LiteralInt

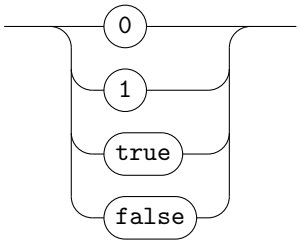


Durch das obenstehende Syntaxdiagrammsystem erhalten Sie die Möglichkeiten für **Literale vom Typ** `int`. Benutzen Sie Literale anderer Typen wie beispielsweise `unsigned int` oder `signed long int`, so müssen sie entsprechende **Suffixe** wie in folgender Tabelle verwenden. Groß- und Kleinschreibung der Suffixe sind gleichbedeutend. Statt U, UL, ULL, L, LL dürfen auch u, ul, ull, l, ll verwendet werden.

	`int`	`long`	`long long`
signed		L	LL
unsigned	U	UL	ULL

Typ bool

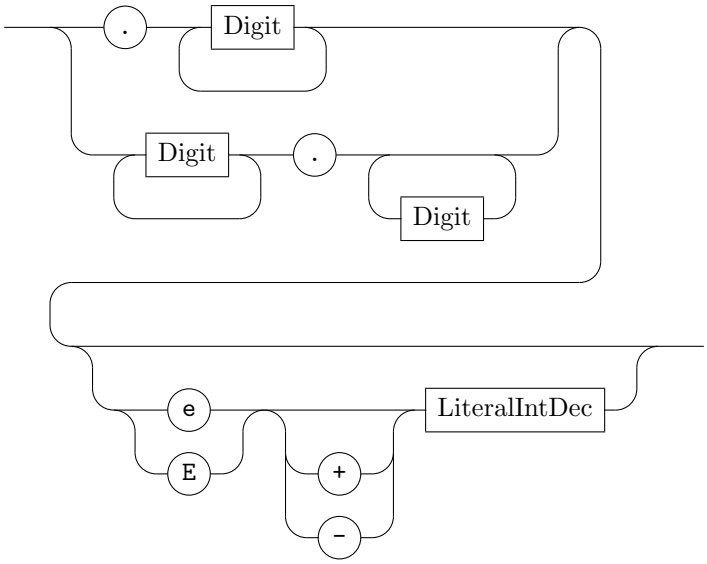
LiteralBool



Eine Variable vom Typ `bool` enthält immer genau den Wert 0 (gleichbedeutend mit `false`) oder 1 (gleichbedeutend mit `true`). Wenn Sie einer solchen Variable eine andere integrale Zahl zuweisen, so wird eine **automatische Konvertierung** nach `bool` durchgeführt. Dabei wird 0 als `false` und jede Zahl ungleich 0 als `true` interpretiert. Für eigene Klassen können Sie selbst eine implizite Konvertierung nach `bool` implementieren.

Fließkommatypen

LiteralDouble



hex floating point

Für Gleitkommazahlliterale benutzt C++ (wie auch C) die **US-Schreibweise mit Dezimalpunkt**. Wie das Synatxdiagramm schon annehmen lässt, sind Fließkommazahlliterale standardmäßig vom Typ `double`. Um Literale der Typen `float` oder `long double` zu erhalten, sind wieder entsprechende Suffixe anzuhängen:

Typ	`float`	`double`	`long double`
Suffix	f oder F		l oder L

Character-Typen

- Zeichen dargestellt als Bitfolge (fester oder variabler Länge)
- Character-Typen sind integrale Typen.
- Zeichensatz / character set / Codetabellen für Zuordnung
- Überladung der Operatoren für die Nutzung als Zeichen

- ASCII Code
  - American Standard Code for Information Interchange
  - 7 Bit Code
  - Nutzung von `char` als Datentyp
  - Stanardlateinalphabet, Zahlen, ...
  - nicht befriedigend, keine landestypischen Zeichen

- ASCII Erweiterungen auf 8 BIT
  - umgangssprachlich auch ANSI Code genannt
  - verschiedene ergänzende Codetabellen
  - ISO 8859-1, ISO 8859-2, ...
  - auch genannt ISO-Latin-1, ISO-Latin-2, ...
  - ISO-Latin-1 für westeuropäische Zeichen
  - keine Abdeckung fernöstlicher Sprachen
  - Beispiel CMD von Windows: alter IBM-PC-Zeichensatz (Codepage 850) (Kompatibilitätsgründe)

- **Fazit: Moderne Systeme verwenden standardmäßig UTF (Unicode Transformation Format)**

- Unicode
  - verschiedene Codierungsverfahren und verschiedene Datentypen
  - `wchar_t` (wide char) schon vor C++11 mit unterschiedlichen Breiten
  - Ein- und Ausgabe von `wchar_t` mit `std::wcin`, `std::wcout`
  - nicht empfohlen bei Anspruch auf Portabilität zwischen Compilern oder Betriebssystemen
  - Encodings (Codierungen) für Unicode:
    - UTF-8
    - UTF-16
    - UTF-32
  - neue Typen `char16_t`, `char32_t` empfohlen

Datentyp	`char`	`wchar_t`	`char16_t`	`char32_t`
Verwendung	ASCII/ANSI, UTF-8	((UTF-16))	UTF-16	UTF-32

LiteralChar

LiteralWideChar

LiteralChar16

LiteralChar32

Zahlwert	Steuerzeichen	Bedeutung
0	\0	Nullzeichen: Markierung vom Ende eines Strings
7	\a	Tonsignal
8	\b	backspace
9	\t	horizontaler Tabulator
10	\n	Zeilenvorschub (neue Zeile)
11	\v	vertikaler Tabulator
12	\f	Seitenvorschub
13	\f	Wagenrücklauf
	\\	Backslash
	\"	doppelte Anführungszeichen
	\'	einfache Anführungszeichen
	\ooo	oktaler Code (ASCII)
	\xhh	hexadezimaler Code (ASCII)

2.2.3 Initialisierung

Mit C++11 wurde eine neue vereinheitlichte Initialisierung eingeführt. Die neue Initialisierungssyntax **verbietet** manche **auto conversion**, inbesondere *Narrowing*, weshalb sie zu bevorzugen ist, um die Fehlererkennung zur Compilezeit zu verbessern. Außerdem existierten vorher (und immer noch) viele verschiedene Initialisierungen für Container, Arrays, ...

Es wird von vielfach empfohlen, neu definierte Variablen immer **sofort** zu **initialisieren**. Andernfalls enthält die neu angelegte Variable einen undefinierten Pseudozufallswert, nämlich den Inhalt des entsprechenden Speicherbereichs seit dem letzten Schreibvorhgang, der dort stattfand.

`Typ Bezeichner1 = 0;`	mit Zuweisungsoperator (schon vor C++11)
`Typ Bezeichner2 {0};`	neue vereinheitlichte Initialisierung
`Typ Bezeichner3 = {0};`	vereinheitlichte Initialisierung mit optionalem '='
`Typ Bezeichner4 {Bezeichner1};`	Der zuzuweisende Ausdruck muss kein Literal sein.

2.2.4 Deklaration und Definition (vereinfacht)

Wenn Sie sich die exakten Definitionen von „Definition“ und „Deklaration“ im C++ Standard anschauen, werden Sie möglicherweise etwas verwirrt sein oder zumindest vor lauter Ausnahmen den „Wald vor Bäumen nicht sehen“. Wir betrachten deshalb hier eine zugegebenermaßen nicht ganz exakte, dafür aber **vereinfachte und intuitivere Darstellung** des Sachverhalts:

Begriff	**Beschreibung**
`Deklaration`	Eine Deklaration (declaration) führt einen Namen ein oder deklariert einen Namen neu (redeclaration) und macht diesen so im betreffenden scope (Sichtbarkeitsbereich) [dem Compiler] bekannt, sodass er dann benutzt werden darf.
`Definition`	Eine Definition ist ein Spezialfall der Deklaration. Intuitiv ausgedrückt ist von einer Definition immer genau dann die Rede, wenn der Compiler explizit angewiesen wird, Speicherplatz für die hinter dem Namen liegende Entität zu reservieren oder der Name „initialisiert“, oder so gesagt *in gewissem Sinne vollständig* spezifiziert wurde.

2.2.5 Einige Operatoren auf primitiven Datentypen

Die Operatoren von C++ werden in einem **gesonderten Kapitel** noch ausführlich behandelt. Hier finden Sie einen kurzen Überblick über **relevante arithmetische Operatoren** und einige Anmerkungen dazu.

binäre arithmentische Operatoren

binärer Operator	**Bedeutung**	**Beispiel**
+	Addition	`int x = 3 + 7; // 10` `long double xx = 23.4L + 43.7L; // 67.1`
-	Subtraktion	`short s = 20 - 21; // -1` `float f = 74.2F - 123.9F; // 49.7`
*	Multiplikation	`long l = 24 * 36; // 864` `double prod = 1.2 * 2.4; // 2.88`

/	Division	`int div = 21 / 6; // 3`
		`double conv = 21 / 6; // 3`
		`double double_div = 21. / 6; // 3.5`

%	Modulo	`int rest = 20 / 6; // 2` **nur auf Ganzzahltypen definiert (!)**

Zu beachten ist insbesondere das Verhalten des `/` -Operators (Divisionsoperator): Wird dieser Operator auf zwei Werten von integralem Typ aufgerufen, so führt dieser eine **Ganzzahldivision** durch. Ist einer der Operanden von einem floating point - Typ, dann führt dieser Operator eine Division mit Nachkommateil durch (floating point Division). Das zweite Beispiel in der Tabelle zur Divisionsoperation zeigt eine Ganzzahldivision und eine anschließende **automatische Konvertierung** von `int` zu `double`.

Alle Operatoren von C++ sind in einer Prioritätsreihenfolge geordnet. Das ermöglicht unter anderem Punkt- vor Strichrechnung. Wollen Sie dagegen eine andere Reihenfolge erzwingen, besteht die Möglichkeit der Klammerung.

Punkt vor Strich: implizite Klammerung:	`int x = 5 + 3 * 7; // 26` `int x = 5 + (3 * 7); // 26`
mit expliziter Klammerung:	`int x = (5 + 3) * 7; // 56`

Zuweisungsoperator und erweiterer arithmetischer Zuweisungsoperator

Operator	Beispiel	Bedeutung
`=`	`x = 3`	Zuweisungsoperator
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 3`	`x = x * 3`
`/=`	`x /= 3`	`x = x / 3`
`%=`	`x %= 3`	`x = x % 3`

2.3 Konvertierungen von Typen

Narrowing und Promotion

2.4 Casts

Die müssen hier aus diesem Kapitel weg!!!

2.5 Zusammengesetzte und konstruierte Datentypen

konstruierter Typ	**Typ-Syntax**	**Deklaration eines Bezeichners**	
		Deklaration (allg.)	Initialisierung (Bsp.)
[identischer Typ]	`T`	`T t;`	`int x {3};`
Pointer auf `T`	`T*`	`T* tptr;`	`int* xptr {&x};`
Referenz auf `T`	`T&`	`T& tref;`	`int& xref {x};`
`N`-Array von `T`, (`N`=0,1,..)	`T [N]` (*nur bei* ***using***)	`T tarray[N];`	`int a [2] {4, xref};`

2.5.1 Pointer

Ein Pointer auf ein Datum (Objekt), auch genannt Zeiger auf das Datum bezeichnet die **Speicheradresse** des Datums. In diesem Sinne „zeigt“ ein Pointer mit seinem Wert, der Speicheradresse, genauer gesagt *der Anfangsspeicheradresse* auf das Objekt.

Pointer haben unabhängig des Typs des Objektes, auf welches sie zeigen, immer dieselbe Größe, welche sich nach der Adressbreite auf Ihrem System richtet.

So gilt zum Beispiel `sizeof(int*) == sizeof(std::vector<double>*)`

Pointer sollten stets entweder mit einer gültigen Adresse auf ein Objekt verweisen oder den **Nullzeiger** enthalten, in C++ `nullptr`. Damit wird markiert, dass der Zeiger momentan auf kein Objekt zeigt.

Pointertyp

Sei `T` ein Typ. Dann ist auch `T*` ein Typ, nämlich der Typ „Pointer auf ein Objekt vom Typ `T`“.

Adressoperator und Dereferenzierungsoperator (Indirektionsoperator)

Um den Pointer auf ein Objekt `t` vom Typ `T` zu bestimmen, nutzt man den unären Adressoperator (`&`). Dabei bezeichnet `&t` den Pointer auf `t`.

Um mit gegeben Pointer `tp` vom Typ `T*` auf das dahinterliegende Objekt zuzugreifen, gibt es den (unären) Dereferenzierungsoperator (`*`). Der Ausdruck `*tp` ist dann wieder vom Typ `T` und bezeichnet das Objekt (ist eine Referenz auf das Objekt), auf welches mit `tp` gezeigt wird.

Zeigerarithmentik

Pointer sind in C++ streng typisiert, damit es ermöglicht wird, die Operatoren speziell für diese Typen zu überladen.

Operator	Bezeichnung	Verwendung
`++, --`	Inkrement, Dekrement	Speicheradresse wird um die Größe des zugrundeliegenden Typs vergrößert bzw. verkleinert.
`+, -`	Addition (Subtraktion)	Es ist möglich, Zahlen zu Pointern hinzuzuaddieren. Dabei wird der Zeiger (die Speicheradresse) nicht um die angegebene Zahl vergrößert, sondern um die Größe des Typs multipliziert mit der Ganzzahl. `ptr + 2` bezeichnet beispielweise das vom Pointer aus gesehen übernächste Objekt im Speicher, sofern Objekte vom selben Typ lückenlos hintereinander im Speicher abgelegt sind.
`&`	Adressoperator	Referenz auf das Objekt (Identifier) $\mapsto$ Adresse des Objektes
`*`	Indirektionsoperator	Pointer auf das Objekt $\mapsto$ Referenz auf das Objekt

Beispiel zu Pointern

```
#include <iostream>

int main(){
        using namespace std;
        using Pdouble = double*;

        int a = 4;
        int b = a++; // a==5, b==4
        double d {4.7};

        int* pa {&a};
        int* pb {&b};

        if (pa == pb) cout << "Das kann nicht sein.";
        if (*pa == *pb) cout << "Das sollte auch falsch sein.";
        *pa = *pb; // a==b==4
        if (*pa == *pb) cout << "Das ist jetzt richtig so.\n";
        pa = pb; // a zeigt jetzt auf b
        *pa = 10;
        cout << a << " " << b << endl; // 4 10

        Pdouble pd = &d;
        pd = &a; // Compiler Error
        pd = reinterpret_cast<double*>(&a);
}
```

2.5.2 Referenzen

Eine Referenz auf ein Datum (Objekt) bezeichnet ein Objekt selbst. So gesagt legen Sie mit einer Referenz einen zusätzlichen Identifier für ein bereits existierendes Objekt an.

Referenzen sind nichts weiter als syntaktischer Zucker für die Verwendung von Pointern. Sie können einem die Arbeit wesentlich erleichtern, da sie einem die ständige Verwendung der Operatoren `&` und `*` ersparen.

Referenzen können nur einmal (mit einer bereits vorhandenen Referenz) initialisiert werden. Hinterher kann das Objekt, auf welches die Referenz *implizit* zeigt, nicht mehr „ausgetauscht“ werden. D.h. Sie können zwar das Objekt hinter der Referenz ändern, die Referenz aber bleibt eine Referenz auf ebendieses spezielle Objekt und kann nicht zu einer Referenz auf ein anderes Objekt an einer anderen Speicherstelle geändert werden.

Referenztyp

Sei `T` ein Typ. Dann ist auch `T&` ein Typ, nämlich der Typ „Referenz auf ein Objekt vom Typ `T`“.

Beispiel zu Referenzen

```
#include <iostream>

void add_23(int& x){
        x = x + 23;
}

int main(){
        using namespace std;

        int a = 4; // Ein neues Objekt vom Typ int auf dem Stack angelegt, mit 4
initialisiert und eine Referenz "a" auf das Objekt wird im scope der main-Funktion
angelegt.
        int& b = a // Eine Referenz b wird angelegt. a und b zeigen implizit auf
dasselbe Objekt.
        int c = 5;

        b = 12; // a==b==12
        add_23(c); // c==28

        cout << a << endl << b << endl << c << endl;
}
```

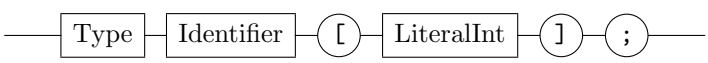
2.5.3 Arrays (C-like)

Unter Arrays (Feldern) versteht man die Zusammenfassung von mehreren Objekten (üblicherweise) gleichen Typs. Um in C++ Arrays zu nutzen, gibt es mehrere Möglichkeiten. Zum einen gibt es die **rohen Arrays**, auch *C-like Arrays* genannt, zum anderen mehrere Containerklassen, insbesondere die Templateklassen **std::vector** und **std::array**.

- **Gründe *für* rohe Arrays**
  - interne Implementierung
  - Gründe der Performance (*naja ...*)
    - mitgebrachte Unsicherheiten $\geq$ bessere Performance
    - C++ von Haus aus schnell. Optimierung nur an kritischen Codestellen sinnvoll
- **Gründe für *Containerklassen* (gegen rohe Arrays)**
  - viele nützliche Features:
  - dynamische Erweiterung (std::vector, nicht std::array)
  - Schutz vor Überläufen (out of range)
  - oft leichter zu handhaben

Array Deklaration

ArrayDeclaration



Mittels *ArrayDeclaration* legen Sie ein Array mit `LiteralInt` vielen Objekten des zugrundeliegenden Typs an. Statt einem *LiteralInt* darf auch ein konkretes *const int* oder eine *constexpr* eingesetzt werden. Entscheidend ist, dass die Größe des Arrays zur Compilezeit bestimmt werden kann.

Erklärung bzw. Ursache: Typen in C++ haben immer eine feste Größe. Typen dynamischer Größe gibt es nicht, jedoch können Typen (*structs, etc ...*) Zeiger enthalten und sich „im Hintergrund“ dynamisch Speicher allokieren.

Deklaration und Initialisierung am Beispiel

Nur-Deklaration	`extern int x[5];`	hier wird noch kein Speicherplatz reserviert, keine Definition
Definition ohne Initialisierung	`int x[5];`	undefinierte Werte im Speicher
leere vereinheitlichte Initialisierung	`int x[5]{}`	alle Einträge mit 0 befüllt
vereinheitlichte Initialisierung (unvollständig)	`int x[5]{1,2,3};`	restliche Einträge mit 0 befüllt. {1,2,3,0,0}
vereinheitlichte Initialisierung	`int x[5] {1,2,3,4,5};`	Elementinitialisierer für alle Einträge
ohne Größe	`int x[] {1,2,3,4};`	implizit `int x[4] {1,2,3,4};`

Indizierungsoperator bzw. Indexoperator

Der Zugriff auf die Elemente erfolgt mit dem Indexoperator `[]` (eckige Klammern). Wurde mit `Typ array[N];` ein Array angelegt, so lässt sich auf diese `N` Elemente mit `array[0], ... , array[N-1]` zugreifen.

Typdefinition

mit typedef	**mit using (*seit C++11*)**
`typedef int INT_5[5];`	`using INT_5 = int[5];`

Es wird empfohlen für Typdefinitionen fürderhin die Variante mit `using` zuu nutzen, da diese den auch Templates gerecht wird.

implizite Konvertierung

Arraytypen können implizit zu Zeigern auf den dahinterliegenden Typ konvertiert werden. Sie erhalten dann einen Zeiger auf das „nullte Element“ des Arrays.

Beispiel zu Arrays

```
    #include <iostream>

    void increment(int* ptr_to_first_element, int count){

    }

    int main(){
            using namespace std;

            int a = 4; // Ein neues Objekt vom Typ int auf dem Stack angelegt, mit 4
initialisiert und eine Referenz "a" auf das Objekt wird im scope der main-Funktion
angelegt.
            int& b = a // Eine Referenz b wird angelegt. a und b zeigen implizit auf
dasselbe Objekt.
            int c = 5;

            b = 12; // a==b==12
            add_23(c); // c==28

            for und range for mit auto&

            cout << a << endl << b << endl << c << endl;
    }
```

2.5.4 Strings

2.5.5 Records

Unter Records versteht man die Zusammenfassung von mehreren Objekten nicht notwendigerweise verschiedenen Typs. Aus C kennt man das Schlüsselwort **`struct`** zum Definieren solcher Typen.

In C++ gibt es zwei Schlüsselwörter zur Definition von Records / Klassen, nämlich `struct` und `class`.

2.5.6 Containerklassen

2.6 Klassen

2.6.1 Konstruktoren

2.6.2 Vererbung

2.6.3 Polymorphie

2.7 typedef und using

2.8 const und constexpr

Mit `const`

2.9 auto

Ein weiteres, sehr nützliches Schlüsselwort, dass Sie spätestens beim Hantieren mit Iteratoren, Template- und Containerklassen nicht mehr vermissen wollen, ist das Schlüsselwort `auto`. Die Bedeutung von `auto` hat sich mit dem Standard C++11 geändert

bis C++03	Angabe einer Speicherklasse Variablen innerhalb von Funktionen automatisch auto, außerhalb nicht erlaubt.
seit C++11	Neudefinition des Schlüsselwortes. Ausdruck einer statischen Typinferenz. automatische Bestimmung des Typs zur Compilezeit. „kleinster Typ“ aus der Initialisierung autonatisch vom Compiler bestimmt (keine dynamische Typisierung)

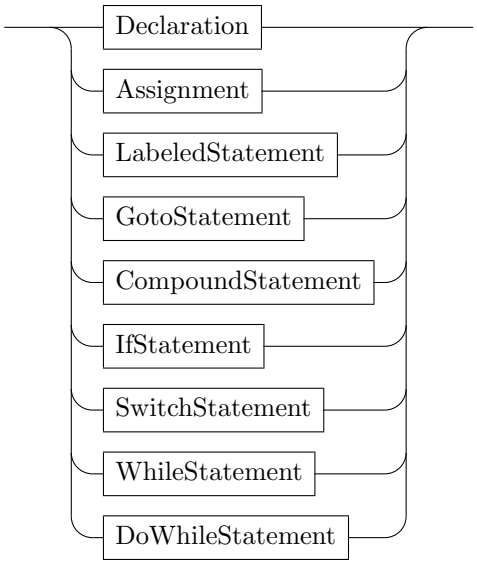
Kapitel 3

Strukturierte Programmierung

3.1 Kontrollstrukturen

In der imperativen Programmierung dreht sich alles mehr oder weniger um - *wie der Name schon sagt* - **imperata**, oder für diejenigen unter Ihnen, die des Latein nicht mächtig sind, *Anweisungen*, auch genannt **Statement***s*.

Statement

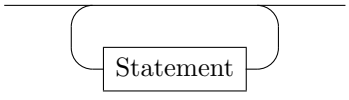


3.1.1 Sequenzen

Grundlegend arbeitet der Prozessor den Maschinencode Anweisung für Anweisung ab und auf diese Weise zuzüglich der darauf basierenden Konstrollstrukturen programmieren Sie auch in C++. Mehrere aufeinanderfolgende Anweisungen nennt man **Sequenz** (*StatementSequence*). Meist treten solche Sequenzen als **Block** innerhalb von geschweiften Klammern auf (*CompoundStatement*). Die Anweisungen einer Sequenz werden genau einmal in ihrer Reihenfolge ausgeführt, wenn die Sequenz einmal ausgeführt wird.

CompoundStatement

StatementSequence



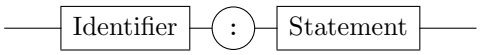
3.1.2 Sprünge

Sprünge gehören eigentlich nicht in die strukturierte, sondern eher in die *unstrukturierte Programmierung*. Dennoch soll hier auf sie eingegangen werden, da sie in gewisser Weise eine Grundlage für höhere Strukturen (wie zum Beispiel **switch** (Abschnitt 3.1.4)) sind und sich manchmal als ganz nützlich erweisen, obgleich man sie gerade wegen des Gegensatzes zu strukturierter Programmierung nur dezent einsetzen sollte.

Sprungmarke

Um Sprünge zu realisieren werden an den einzelnen **Zielpositionen** Sprungmarken benötigt. Diese (*also die genutzten Identifier*) müssen zuvor in keiner Weise deklariert werden, sondern können einfach an Ort und Stelle gesetzt werden.

LabeledStatement



Mittel angegebener Syntax legen Sie eine Sprungmarke vor eine Anweisung. Springen Sie später zu dieser Sprungmarke, so springen Sie **vor die nächste Anweisung**, ab welcher nach dem Sprung der sequenzielle Programmablauf fortgesetzt wird.

Sprunganweisung

Nachdem Sie eine Sprungmarke gesetzt haben, können Sie mit der folgenden Syntax an beliebiger Codeposition einen Sprung dorthin vollziehen:

GotoStatement

Regeln beim Umgang mit goto

Hier seien einige Bedingungen zusammengefasst, die **hinreichend** sind, um keine Fehler im späteren Programmablauf zu riskieren.

- nicht in andere CompoundStatements hineinspringen
  - Insbesondere sollte man nie in einen Schleifenrumpf hineinspringen, nur hinaus.
  - Auch Sprünge in then- bzw. else-Klauseln sollten nicht vollzogen werden. (*Das verzeiht einem der Compiler allerdings noch eher als ein Sprung in eine Schleife*)
  - Man sollte insbesondere nicht zwischen then- und else-Zweig eines IfStatement hin- oder herspringen.
  - Man sollte erst recht nicht in andere Funktionen springen.
- Wenn Rümpfe von (if, while, ...)-Konstrukten nicht direkt ein CompoundStatement darstellen, sollte man auch dann nicht dort hineinspringen.
- Man sollte keine Deklarationen / Initialisierungen überspringen
  - Nichtbefolgen könnte gerade bei der Initialisierung eines Objektes fatal sein, welches dynamisch Speicher allokiert.

`http://www2.informatik.uni-halle.de/lehre/c/c35.html`

Sprünge im weitergefassten Sinne	
`break;`	Sprung aus (nur) innerster Schleife hinaus
`continue;`	Sprung zum nächsten Iterationsschritt der Schleife (Sprung an das Ende des Schleifenrumpfes)
`return` *`[optional: Expression]`*`;`	Beendigung der aktuellen Funktion [mit Rückgabewert]
`exit (`*`int Expression`*`)`	Beenden des Programms mit angegebenem exit-code.
`abort;`	abnormale Beendigung des Programms.

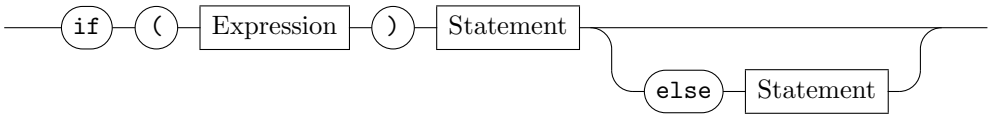
3.1.3 if - Verzweigung

Erläuterung

Die Verzweigung mit `if` realisiert die bedingte Ausführung von Programmcode. `Expression` muss einen nach `bool` konvertierbaren Typ aufweisen.

Syntaxdiagramm

IfStatement



Verhalten

Wenn Sie **if** schachteln, dann wird der optionale `else`-Zweig immer an die innerste Verzweigung assoziiert.

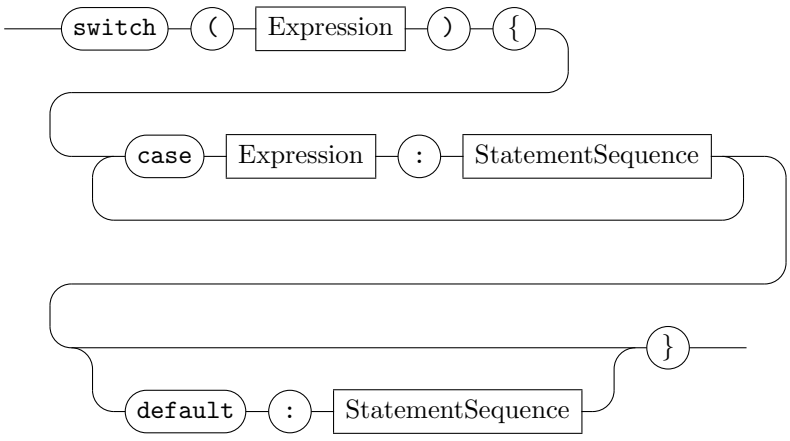
Wenn *Expression* zu `true` auswertet, wird das *Statement* direkt hinter der Bedingung ausgeführt. Andernfalls wird das *Statement* hinter `else` ausgeführt, sofern vorhanden.

3.1.4 switch - Mehrfachverzweigung

Mit der `switch`-Anweisung lässt sich eine Fallunterscheidung, auch genannt Mehrfachverzweigung implementieren. Die `switch`-Syntax ist dabei nur **syntactic sugar** (syntaktischer Zucker) für die Verwendung von einem geschachtelten `if` in Kombination mit `goto` und Sprungmarken. Dabei ist der Ausdruck, nach welchem verzweigt wird, jedoch auf integrale Typen (einschließlich `bool` und enum-Typen) beschränkt.

Synatxdiagramm

SwitchStatement



Verhalten

Expression wird ausgewertet und anschließend wird hinter das `case` mit dem entsprechenden Wert gesprungen. Ist der Wert von *Expression* nicht als `case` enthalten, so wird sofern vorhanden zu `default` gesprungen. `Achtung!` Es wird zur entsprechenden Sprungmarke gesprungen und es werden von da an alle Anweisungen des Blockes (CompoundStatement) sequenziell abgearbeitet. Insbesondere heißt dies, dass im allgemeinen Fall auch alle Anweisungen des nächsten Sprunglabels abgearbeitet werden. Wollen Sie dies unterbinden, so können Sie mit `break` aus dem `switch`-Block hinausspringen.

vergleich elif #

`http://en.cppreference.com/w/cpp/language/switch`

3.1.5 while

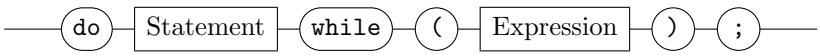
break, continue #

WhileStatement

`http://en.cppreference.com/w/cpp/language/while`

3.1.6 do while

DoWhileStatement



`http://en.cppreference.com/w/cpp/language/do`

3.1.7 for (klassisch)

wird ergänzt

3.1.8 for (foreach)

3.2 Funktionen

3.3 Operatoren

3.4 Modularisierung

Kapitel 4

Zusätzliche Features

4.1 Templates

4.2 Exceptions

4.3 Multithreading

Kapitel 5

Ein Blick in Bibliotheken

5.1 Wertebereiche mit ¡limit¿