

Praxis der Programmierung

Von C zu C++

Objektorientierte Programmierung mit C++

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Einführung in C++

Weiterentwicklung von C

- C++ übernimmt die Konzepte von C:
 - Compilerarchitektur (Präprozessor, Compiler, Linker)
 - Header-Dateien, (stark erweiterte) Standardbibliothek
 - Variablen, Pointer, Datentypen (einschl. Arrays, Strukturen, Unionen, ...)
 - Kontrollstrukturen, Funktionen, Konstanten
 - Kommentare, ...
- weitere Datentypen (bool, erweiterte Zeichensätze, z.B. wchar_t)
- Namensräume
- Unterstützung des objektorientierten Paradigmas
- Templates
- geringe syntaktische Varianten

Bibliotheken

- Einbinden mit `#include <...>` (ohne Endung `.h`)
- sehr mächtige Bibliothek mit Lösungen im **C++**-Standard
- **C**-Bibliotheken vorhanden:
`<name.h> ↔ <cname>` , z.B. `<cstdio>`, `<cstdlib>`, ...
- Ein- und Ausgabe mit `<iostream>`
 - Erlaubt Lenken von Datenströmen in Ein- und Ausgabeobjekte
 - `cout` ist Standardausgabe
 - `cerr` ist Standardfehlerausgabe
 - `cin` ist Standardeingabe
 - `cout`, `cin`, `cerr` im **Namensraum `std`** definiert

Das Hello World-Beispiel

```
/* Programmname.cpp
 * Quellcodedateien enden auf .cpp
 */

#include <iostream>           // Praeprocessoranweisung
using namespace std;        // Anweisung: Benutzen des Namensraums std

int main() {

    cout << "Hello World!"; // << fuer Umlenkung des Datenstroms
    cout << endl;           // endl ist Konstante fuer das Zeilenende in std

    // alternatives einzeiliges Kommando:
    cout << "Hello World!" << endl;

} // fehlendes return-Kommando bei main() wird vom Compiler toleriert
```

Namensräume

- definieren Bereiche, in denen Namen / Bezeichner eindeutig sein müssen
- in verschiedenen Namensräumen kann der gleiche Name verwendet werden
- *Beispiele:*
 - Telefonnummern in Vorwahlbereichen: 0331 123456
030 123456
 - Telefonnummern mit Vorwahl in Ländernetzen: +49 30 123456
+36 30 123456
 - Dateinamen in Ordnern; diese in übergeordneten Ordnern, ...
 - in Netzwerken absolute Pfadnamen auf Hosts (Rechnernamen), ...

Qualifizierte Namen

- **unqualifizierte Namen:** die Bezeichner selbst

Beispiel: meineDatei

- **qualifizierte Namen:** mit Angabe des Namensraums

Beispiel: /home/rlehre/meineDatei

- in **C++:** namensraum::bezeichner

Beispiel: std::cout, std::cin, std::endl

- Namensraum `std` enthält Bezeichner der Standardbibliothek
- Definition eigener Namensräume möglich ... *später*

Das Hello World-Beispiel (2)

```
/* Programmname.cpp
 * Quellcodedateien enden auf .cpp
 */

#include <iostream>          // Praeprozessoranweisung
/* using namespace std; auskommentiert */

int main() {

    std::cout << "Hello World!"; // << fuer Umlenkung des Datenstroms
    std::cout << std::endl;      // endl ist Konstante fuer das Zeilenende in std

    // alternatives einzeiliges Kommando:
    std::cout << "Hello World!" << std::endl;

} // Compiler c++ oder g++, z.B. g++ -Wall Hello.cpp -o Hello
```

Ein- und Ausgabe

Formatierte Ausgabe

- Verwendung von Funktionen aus `<cstdio>`
~> z.B. bei formatierter Ausgabe von Zahlen
- Verwendung von **Manipulatoren**, z.B.:
 - `endl` (`<iostream>`)
~> erzwingt Ausgabe der bisher eingegebenen Zeile und einen Zeilenvorschub
 - `setw(breite)` (`<iomanip>`)
~> rechtsbündige Ausgabe des nächsten Ausgabestroms und Auffüllen mit Leerzeichen auf `breite`
 - `setfill(chr)` (`<iomanip>`)
~> ab jetzt Ersetzen des Leerzeichens zum Auffüllen durch `chr`
 - `left` und `right` (`<iostream>`)
~> ab jetzt Umstellen auf Links- bzw. Rechtsbündigkeit

Beispiel Verwendung von Manipulatoren

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

int main() {
    cout << setw(7) << "42" << endl;
    cout << setfill('-');
    cout << setw(7) << "42" << endl;
    cout << left << setw(7) << "42" << endl;
}
```

erzeugt die Ausgabe

```
      42
-----42
42-----
```

Weitere Manipulatoren

- `dec, oct, hex (<iostream>)`
↪ ab jetzt Darstellung ganzer Zahlen als Dezimal-, Oktal- bzw. Hexadezimalzahl
- `showbase (<iostream>)`
↪ ab jetzt Zeigen von 0x bei Hexadezimalzahlen und 0 bei Oktalzahlen
- `scientific, fixed (<iostream>)`
↪ ab jetzt in Exponential- bzw. Festkommadarstellung
- `setprecision(n) (<iomanip>)`
↪ ab jetzt n Nachkommastellen

Elementare Datentypen

Der Datentyp `bool`

- Literale `true` und `false`
- Ausgabe als `1` bzw. `0`
Umstellung mit den `iostream`-Manipulatoren `boolalpha` und `noboolalpha`
- Boolesche Ausdrücke durch Vergleich arithmetischer Ausdrücke, (`a < b+2`)
und durch Funktionen mit Rückgabotyp `bool`
- logische Verknüpfung mit `!` (Negation), `&&` (Und), `||` (Oder)

Zeichentypen

- `char` (1 Byte für ASCII-Zeichen)
- `wchar_t` (internationaler Zeichensatz)
- `char16_t` (16-Bit UNICODE) im **C++11-Standard**
- `char32_t` (32-Bit UNICODE) im **C++11-Standard**
(schließt auch asiatische Zeichen ein)

Übersetzung im C++11-Standard mit `g++ -std=c++11`

Einige syntaktische Varianten

- **Initialisierung von Variablen**

1. `int zahl = 0;` // wie in C
2. `int zahl(0);` // wie Funktionsaufruf
3. `int zahl{0};` // direkte Initialisierung im C++11-Standard
4. `int zahl = {0};` // direkte Initialisierung im C++11-Standard

- **Typkonvertierung**

1. `int n = 200;`
 `(char) n` // wie in C
2. `char(n)` // wie Funktionsaufruf

Das Paradigma der objektorientierten Programmierung

Grundidee des Paradigmas

- *bisher*: **Wie** soll ein Ziel erreicht werden?
 - Algorithmen/Prozeduren im Mittelpunkt
 - Daten werden durch Prozeduren manipuliert
 - ↔ prozedurale Programmierung / imperative Programmierung
- *jetzt*: **Was** soll erreicht werden?
 - Daten werden zu Typen zusammengefasst (siehe struct-Typen)
 - Durch Zuweisung von Werten (konkreten Daten) entstehen Datenobjekte.
 - Algorithmen durch Aktivitäten / Interaktionen von Datenobjekten

Objekte

- repräsentieren Objekt der realen Welt in der Terminologie objektorientierter Programmiersprachen
- besitzen einen **Namen**, mit dem sie angesprochen werden können
- besitzen **Attribute** (Eigenschaften), deren Werte im Allgemeinen veränderlich sind und den **Zustand** der Objekte bestimmen
- reagieren auf an sie gesendete **Botschaften** durch gewisse Aktionen

Klassen

- Gesamtheiten (Mengen) von Objekten
 - mit denselben Attributen,
 - die dieselben Botschaften verstehen und auf dieselbe Weise darauf reagieren,
 - unterscheiden sich in den Werten ihrer Attribute
 - Verhalten eines Objekts kann von seinem Zustand abhängen
- Objekte sind *Exemplare (Instanzen)* einer Klasse
 - alle Attribute besitzen einen Wert (Objektzustand)
 - repräsentiert durch eine Variable vom **Typ** der Klasse,
z.B. *Exemplar* obj *der Klasse* C1s:

```
C1s obj;
```

Klassen *versus* Strukturen

```
typedef struct datum {  
    int day;  
    int month;  
    int year;  
} Date;
```

```
class Date {  
public:  
    int day;  
    int month;  
    int year;  
};
```

- definieren den gleichen **Datentyp**
- Membervariablen in Strukturen sind Attribute in Klassen (**Datenelemente**)
- Schlüsselwort `public` erlaubt Zugriffe auf Attribute wie auf Membervariablen bei Strukturen
- direkte Initialisierung erst ab **C++11**-Standard möglich

Klassen und Zugriffe auf Datenelemente

```
class Date {  
public:  
    int day;  
    int month;  
    int year;  
};
```

```
int main() {  
    Date heute;  
    heute.day = 19;  
    heute.month = 6;  
    heute.year = 2015;  
}
```

Verhalten von Objekten: Methoden

- Definition von Funktionen in der Klassendef. (innerhalb von `class {...}`)
- Signaturen der Methoden definieren, welche “Botschaften” von Objekten dieser Klasse verstanden werden
- Implementierungen definieren Verhalten/Reaktion der Objekte

```
class Date {
public:
    int day, month, year;

    void setNewYear(int y) {
        day = 1;
        month = 1;
        year = y;
    }
};
```

```
int main() {
    Date d;
    d.setNewYear(2016);
    Date * dptr = &d;
    dptr->setNewYear(2016);

    // beides liefert 1.1.2016;
}
```

Auflösung von Verdeckungen

- lokale Variablen (auch Parameter) verdecken gleichnamige Datenelemente
- Nutzen der vordefinierten Selbstreferenz `this` (ist Pointer!)
↪ es soll explizit ein Element des Objekts angesprochen werden

```
class Date {  
public:  
    int day, month, year;  
  
    void setNewYear(int year) {  
        day = 1;  
        month = 1;  
        this->year = year;  
    }  
};
```

```
int main() {  
    Date d;  
    d.setNewYear(2016);  
    Date * dptr = &d;  
    dptr->setNewYear(2016);  
  
    // beides liefert 1.1.2016;  
}
```

Konzepte des objektorientierten Paradigmas

- Klasse und Objekt
- Datenelemente/Instanzvariablen (Attribute) und Methoden
- Kapselung
Interna von Objekten sind nach außen unsichtbar und können von außen nicht manipuliert werden
- Vererbung
Weitergabe von Merkmalen und Fähigkeiten (Datenelementen und Methoden)
→ hierarchisches Klassensystem (*Ober-* und *Unterklassen*)
- Polymorphismus
verschiedene Reaktionen von Instanzen verschiedener Unterklassen auf eine gemeinsam verstandene Botschaft
→ Überschreiben von Methoden

Erzeugen, Verwenden und Zerstören von Objekten

Initialisieren von Objekten

```
class Date {  
public:  
    int day;  
    int month;  
    int year;  
};  
  
int main() {  
    Date heute;  
    heute.day = 19;  
    heute.month = 6;  
    heute.year = 2015;  
}
```

unsauber, da Initialisieren aller Datenelemente nicht erzwungen ist:

Initialisieren von Objekten

```
class Date {
public:
    int day;
    int month;
    int year;
};

int main() {
    Date heute;
    heute.day = 19;
    heute.month = 6;
    heute.year = 2015;
}
```

unsauber, da Initialisieren aller Datenelemente nicht erzwungen ist:

```
// weiter in main:

    Date morgen;
    morgen.day = heute.day + 1;
    morgen.month = heute.month;

// Ausgabe aller Attributwerte von morgen ergibt etwas wie 20.6.4197168
```

Konstruktoren

- Definition einer speziellen Methode zum Erzeugen eines Objekts (**Konstruktor**)
- Name des Konstruktors ist immer Name der Klasse
- kein Rückgabetyt (*auch nicht void*)

```
class Date {  
    public:  
        int day, month, year;  
  
        Date() { // hier parameterlos  
            day = 1; month = 1; year = 1;  
        }  
};
```

- Programmierer muss für Initialisierung aller Datenelemente sorgen!!!

Überladen von Konstruktoren

- Vereinbarung mehrerer Konstruktoren mit unterschiedlichen Parameterlisten
- **Standardkonstruktor**
 - parameterlos \rightsquigarrow genau einer pro Klasse (`Date()`)
 - Initialisierung aller Datenelemente mit festen Standardwerten
- **Initialisierungskonstruktor**
 - Initialisierung (einiger) Datenelemente mit Parameterwerten
(`Date(int year)`)
 - verschiedene Parameterlisten für verschiedene Initialisierungen
- Compiler erkennt aufgerufene Funktion am Namen und den aktuellen Parametern

Verwenden von Konstruktoren

- Definition einer Objektvariablen ruft Konstruktor auf:

```
Date d;           // Standardkonstruktor Date()  
Date d(2014);     // Initialisierungskonstruktor Date(int year)
```

- ist *gar kein* Konstruktor definiert, so wird der *implizite* Konstruktor aufgerufen
- Der **implizite Konstruktor** ist parameterlos und erzeugt eine Instanz, ohne die Datenelemente explizit zu initialisieren; (z.B., wenn Initialisierung durch Methoden erfolgen soll)
- impliziter Konstruktor kann als Standardkonstruktor angefordert werden:

```
Date() = default;
```

Objekte ohne impliziten Konstruktor anlegen

Date d; nur möglich, wenn ein parameterloser Konstruktor zur Verfügung steht

```
class Date {
public:
    int day, month, year;
    Date(int year) {
        day = 1;
        month = 1;
        this->year = year;
    }
};
```

```
int main() {
    Date gehtNicht;          // Compilerfehler
    Date gehtDoch(2014);    // Aufruf des definierten Konstruktors
}
```

Überladen von Konstruktoren bei Date

```
class Date {
public:
    int day, month, year;
    Date() {
        day = 1; month = 1; year = 2000;
    } // Standardkonstruktor

    Date(int year) {
        day = 1;
        month = 1;
        this->year = year;
    } // ein Initialisierungskonstruktor

    Date(int day, int month, int year) {
        ... } // noch ein Initialisierungskonstruktor
};
```

Erzeugen von Objekten auf dem Heap

- Anlegen eines Objekts im Speicher mit `new`
- liefert Referenz auf ein vorinitialisiertes Objekt auf dem Heap

```
Date * irgendwann;  
irgendwann = new Date;
```

```
cout << (*irgendwann).day << "." (*irgendwann).month << ".";  
cout << irgendwann->year << endl;
```

- `new` kann Aufruf von `malloc()` "ersetzen"

Zerstören von Objekten

- bei “manueller” Definition eines Objekts (Variablendefinition):
Benutzen des Stacks
↔ Speicherfreigabe bei Verlassen des Blocks mit der Variablendefinition
- bei Anlegen des Objekts mit `new`:
Benutzen des Heaps
↔ Speicherfreigabe durch Programmierer mit `delete` erforderlich:

```
delete irgendwann;  
irgendwann = 0;           // irgendwann = nullptr;  
                           // im C++11-Standard
```

Destruktoren

- werden automatisch bei Zerstören von Objekten aufgerufen (bei `delete` bzw. Erreichen des Blockendes, in dem das Objekt definiert wurde)
- impliziter Standarddestruktor, falls kein Destruktor definiert ist
- Definition: `~Klassenname()`, z.B. `~Date()`
- wichtig zum Freigeben von Ressourcen, die das Objekt angefordert hat (z.B. durch Erzeugen von Exemplaren anderer Klassen, die nur von diesem Objekt genutzt wurden)

Destruktoren Beispiel

```
class HighScore {  
public:  
    int score;  
    Date * d;  
    HighScore();  
    ~HighScore();  
};
```

```
HighScore::HighScore() {  
    score = 0;  
    d = new Date;  
}
```

```
HighScore::~~HighScore() {  
    delete d;  
    d = 0;  
}
```

Übersichtliche Klassendefinitionen

... durch Vorwärtsdeklaration der Konstruktoren und Methoden:

```
class Date {  
public:  
    int day, month, year;  
    Date(int year);  
    void setNewYear(int year);  
};
```

```
Date::Date(int year) {  
    this->year = year;  
}
```

```
void Date::setNewYear(int year) {  
    day = 1; month = 1;  
    this->year = year;  
}
```

```
/* Die Namen muessen  
 * qualifiziert werden:  
 * Date::name  
 *  
 * Dadurch werden sie  
 * von global definierten  
 * Namen (wie Date) unterschieden.  
 */
```

Kapselung

Kapselung

- Klassenelemente können vor dem Zugriff von außen geschützt werden
- Schlüsselwort `private` (ist default-Einstellung)
 - ↪ nur Methoden der Klasse selbst können zugreifen/aufrufen
- Schlüsselwort `public`
 - ↪ alle Funktionen können zugreifen
- Schlüsselwort `friend` vor fremden Klassen oder Methoden
 - ↪ diese dürfen auf `private` Elemente zugreifen

Kapselung bei Date

```
class Date {
    int day, month, yaer;    // private als default-Modifikator
public:
    Date();
    ~Date();
    int getDay();           // Getter-Methoden,
    int getMonth();         // damit die Datenelemente
    int getYear();          // gelesen werden koennen

    void setDay(int day);   // Setter-Methoden,
    void setMonth(int month); // damit die Datenelemente
    void setYear(int year); // veraendert werden koennen
};
```

Ausnutzung der Kapselung

- Unterscheiden von Lese- und Schreibzugriffen durch gezielte Definition von Gettern und Settern
- Kontrolle über die Art der Zugriffe:

```
int Date::getDay() {  
    return day;  
}
```

```
void Date::setDay(int day) {  
    if (0 < day && day < 32)  
        this->day = day;  
}
```

Gezieltes Öffnen der Kapselung

Erlauben von Zugriffen auf private Datenelemente für eine globale Funktion:

```
class Date {
public: Date();
private: int day, month, year;
friend void setNewYear(Date * d, int year);
};

Date::Date() { day = 1; month = 1; year = 2000; }

void setNewYear(Date * d, int year) { d->year = year; }

int main() {
    Date nextNewYear; // Aufruf des Standardkonstruktors
    setNewYear(&nextNewYear, 2016);
}
```

Befreundete Klassen

- Erlauben des Zugriffs auf private Elemente für *alle* Methoden einer Klasse

```
class YourClass {  
    friend class YourOtherClass; // YourOtherClass als friend deklariert  
    private:  
        int topSecret;  
};
```

```
class YourOtherClass {  
    public:  
        void change( YourClass yc );  
};
```

```
void YourOtherClass::change( YourClass yc ) {  
    yc.topSecret++; // Zugriff ist erlaubt  
}
```

Namenskonventionen

- Syntax: Bezeichner wie in **C**
- sprechende Bezeichner (Ausnahme: Schleifenzähler u.ä.)
- Grundsätze für Bezeichner:
 - einfache Datentypen vollständig klein `int`
 - Klassen jedes Teilwortes groß `Date`
 - Variablen und Methoden große Anfangsbuchstaben
 außer beim ersten Teilwort `ptr`
 `setNewYear`
 - Konstanten vollständig groß `PI`