Programmieren / Algorithmen & Datenstrukturen

Grundlagen (i), Teil 7

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Grundlagen (i)

Inhalt.

- Hallo C++
- Objekte, Typen, Werte, und Steuerungsprimitive
- Berechnungen und Anweisungen
- Fehler
- Fallstudie: Taschenrechner
- Funktionen und Programmstruktur
- Klassen

Klassen

Einige technische Details zu C++ Klassen.

- Wir werden uns noch etwas genauer mit den technischen Einzelheiten von *benutzerdefinierten Typen* (v.a. Klassen) in der Programmiersprache C++ befassen.
- Wir werden am Beispiel einer Datumsklasse vorgehen.
- Sie erinnern sich:
  - Es geht nicht um eine auch nur annähernd vollständige Beschreibung der Syntax und Semantik von C++.
    - Mehr als 1000 Seiten C++ Sprachbeschreibung...
  - Unser Ziel: hohe Verständlichkeit und hoher Nutzen im Verhältnis zum Aufwand.
  - Unser Hauptinteresse: Programmieren, Algorithmen & Datenstrukturen.
  - Die Programmiersprache: unser Werkzeug.

Benutzerdefinierte Typen

Typen, die nicht wie `int` oder `double` in C++ eingebaut sind, werden benutzerdefinierte Typen genannt (user-defined types, UDTs).

- Zu den benutzerdefinierten Typen gehören die Typen der StdLib wie etwa `string` oder `vector<T>`.
  - Die Typen der StdLib sind, ebenso wie die integrierten Typen, ein fester Bestandteil von C++, sie sind im Sprachstandard normiert.
  - Sie sind dennoch benutzerdefiniert, weil sie mithilfe derselben Sprachmittel und mit denselben Techniken erstellt sind, wie die Typen, die man selbst schreibt.
    - Wie etwa `X_div` oder `Token_stream` aus dieser Lehrveranstaltung.
  - D.h. den Autoren der StdLib standen für die Arbeit weder spezielle Privilegien noch irgendwelche Tricks zur Verfügung, die nicht auch jeder andere nutzen könnte.
- Warum definieren wir eigene Typen?
  - Unsere eigenen Typen helfen uns, unsere Programmideen direkt als Quellcode darzustellen.
  - Da alle möglichen, unvorhersehbaren Programme notwendig sein könnten, ist es ist unmöglich, alle nützlichen Typen im Voraus zu kennen und sie in einer Bibliothek zur Verfügung zu stellen.

Benutzerdefinierte Typen

Es gibt zwei Arten von benutzerdefinierten Typen: Klassen und Aufzählungen.

- Klassen
  - Schlüsselwort `class` (Sonderfall `struct`).
  - In C++ neben Funktionen der wichtigste Baustein für die Konstruktion von Programmen.
  - Wenn Sie etwas als separate Einheit für ein Programm wahrnehmen, ist es vermutlich sinnvoll, eine Klasse dafür zu definieren.
- Aufzählungen
  - Schlüsselwort `enum`.
  - Ein sehr einfacher benutzerdefinierter Typ, der eine Reihe von Werten als symbolische Konstanten (sog. Enumeratoren) definiert.

Aufzählungen

Es gibt zwei Arten von benutzerdefinierten Typen: Klassen und Aufzählungen.

- Durch das Schlüsselwort `enum` definiert man in C++ eine *Aufzählung*.
  - D.h. eine Liste von symbolischen Konstanten (diese werden *Enumeratoren* der Aufzählung genannt).
- Eine Aufzählung kann benannt sein oder auch anonym bleiben.
- Eine Aufzählung besteht einfach aus der Liste ihrer Enumeratoren und kann eingesetzt werden, wenn zusammenhängende symbolische Konstanten gebraucht werden.

```
enum Month { jan=1, feb=2, mrz=3, apr=4, mai=5, jun=6,
             jul=7, aug=8, sep=9, okt=10, nov=11, dez=12 };

Month m { feb };
```

- Praxistipp: verwenden Sie `enum` *nicht* im globalen Scope.
  - Grund: Aufzählungen definieren keinen eigenen Gültigkeitsbereich, d.h. die Enumeratoren liegen im selben Scope wie die Aufzählung selbst.
  - (Die sog. `enum class` hat jedoch einen eigenen Scope, dieses Sprachkonstrukt wird im Rahmen dieser Lehrveranstaltung aber nicht weiter behandelt.)

Aufzählungen

Eine Aufzählung (Schlüsselwort `enum`) ist ein sehr einfacher, benutzerdefinierter Typ.

- Eine Aufzählung besteht aus einer Liste von Enumeratoren, die eine Menge von Werten definieren.
- Beispiel

Aufzählung namens `Month`

```
enum Month {
    Jan=1, Feb, Mrz, Apr, Mai, Jun,
    Jul, Aug, Sep, Okt, Nov, Dez
};
```

Enumerator
namens `Jun`
vom Aufzählungstyp `Month`
mit Wert 6

```
Month m{Mrz};
int i{m}; // OK, man kann den numerischen Wert eines Month Objekts erhalten.
m = 7; // Fehler: Typ int kann nicht an Typ Month zugewiesen werden.
Month mm{ Month( 5 ) }; // OK, int nach Month konvertieren (ohne Pruefung!)
```

- Die Aufzählung wird manchmal auch als Enumeration bezeichnet, ein Begriff, der leicht mit Enumerator verwechselt werden kann.
  - Engl. enumeration vs enumerator.

Aufzählungen

Anwendung von `enum` anhand von Beispielen.

- Eine einfache Liste von symbolischen Konstanten.

```
enum { red, blue }; // Anmerkung: enum {} bestimmt keinen Gueltigkeitsbereich.
int z{ blue }; // blue ist gueltig
enum { red, yellow, green }; // Fehler: red ist schon vorher definiert worden.
```

- Ein benannter Typ mit einer Liste von symbolischen Konstanten.

```
enum Color { red, blue, green /*…*/ };
enum Month { Jan, Feb, Mrz /*…*/ };
Month m1{Jan};
Month m2{red}; // Fehler, red ist nicht vom Typ Month.
Month m3{1};   // Fehler, 1 ist int-Literal, nicht vom Typ Month.
int i{m1};     // OK, die Month Variable wird implizit
               // in ihren numerischen Wert 0 umgewandelt.
```

- Aufzählungen besitzen *keinen* eigenen Scope, d.h. jeder Enumerator liegt in demselben Scope wie seine Aufzählung.
  - Folglich sollten Aufzählungen in einem möglichst engen Gültigkeitsbereich definiert werden (z.B. innerhalb einer Klasse).
  - Alternativ kann das Konstrukt `enum class` oder `enum struct` verwendet werden (wird in dieser Lehrveranstaltung nicht vertieft).

Aufzählungen

Anwendung von `enum` anhand von Beispielen.

- Standardwerte

```
// Der erste Wert ist 0.
// Der naechste Wert ist "eins mehr als der vorherige Wert".
enum { Hund, Katze, Maus }; // Hund==0, Katze==1, Maus==2
```

- Ausdrückliche Nummerierung

```
enum { jan=1, feb, mrz /* … */ }; // feb==2, mrz==3
enum stream_state { good=1, fail=2, bad=4, eof=8 };
int flags{fail+eof};  // flags==10
stream_state s{flags}; // Fehler: int kann nicht an stream_state zugewiesen werden.
stream_state s2{ stream_state(flags) }; // flags nach stream_state casten.
                                        // Vorsicht: ohne Pruefung!
```

- Bemerkungen
  - Mehrere Enumeratoren können den selben numerischen Wert repräsentieren.
  - Ein Objekt vom Aufzählungstyp kann einen numerischen Wert enthalten, dem kein Enumerator entspricht.
  - Man kann für `enum` (und `enum class` und `enum struct`) keine Konstruktoren definieren.

Klassen und Klassenmember

Eine Klasse enthält null oder mehr Elemente, die auch als Member bezeichnet werden.

- Klassen beschreiben benutzerdefinierte Typen.
- Klassen werden vom Benutzer (d.h. vom Programmierer) aus Einzelteilen zusammengesetzt.
- Die Teile, aus denen sich eine Klasse zusammen setzt, bezeichnet man als Member der Klasse.
- Die häufigsten Memberarten sind
  - Datenmember (*Attribute*), die bestimmen, wie die Objekte der Klasse repräsentiert werden,
  - Memberfunktionen (*Methoden*), die die Operationen definieren, die mit Objekten der Klasse ausgeführt werden können,
  - weitere benutzerdefinierte Typen (verschachtelte Klassen), die innerhalb der Klasse liegen.
- Für den Zugriff auf die Member einer Klasse haben wir die *Punktsyntax* kennen gelernt, für die ein Objekt vom Typ der Klasse benötigt wird:

```
objektname.membername
```

Schnittstelle `public`, Implementierung `private`

Auf die Implementierung einer Klasse kann von außerhalb der Klasse nur über die Schnittstelle der Klasse zugegriffen werden ("*Kapselung*").

- Eine Klasse mit Schnittstelle und Implementierung:

```
class X { // eine Klasse namens X
   public: // public-Member
        // Schnittstellen-Elemente, BENUTZERSICHT
        // Die Benutzer sind andere Programmierer
        // Verwendung auch von ausserhalb der Klasse
        // Inhalt: meist Methoden und Typen
        // ggf. Daten (selten, gehoeren i.Allg. zur Implementierung)
   private: // private-Member
        // Implementierungs-Elemente, AUTORENSICHT
        // Die Autoren sind die Programmierer der Klasse
        // Verwendung nur von Membern dieser Klasse
        // Methoden, Typen, Daten
};
```

- Klassenmember sind bei `class` standardmäßig `private`.
  - Ein `struct` ist eine Klasse mit Membern, die standardmäßig `public` sind.
  - Eine Klasse, die *nur* aus `private` Membern besteht, ist für sich normalerweise vermutlich sinnlos.

Eine Datumsklasse als `struct`

Repräsentation eines Datums als Typ.

- Annahme: ein Datum besteht aus Jahr, Monat und Tag.

```
struct Date {    // sehr einfach
 int y;          // Membervariable fuer das Jahr
 int m;          // Membervariable fuer den Monat
 int d;          // Membervariable fuer den Tag
};

Date heute{}; // eine Variable vom Typ Date namens heute

heute.y = 2011;  heute.m = 9;   heute.d = 4;   // OK
heute.y = -1582; heute.m = -10; heute.d = -13; // sinnlos
heute.y = 2400;  heute.m = 2;   heute.d = 29;  // ?
```

- Wenn wir uns einen eigenen Typ (wie `Date`) bauen, fragen wir uns immer: *Welche Operationen werden für diesen Typ benötigt?*
  - Zunächst und vor allem: ein Datum *gültig* initialisieren, etwas naiv etwa durch: `void initDate( int y, int m, int d ) { /* … */ }`
  - Aber auch z.B. ein Datum korrekt verändern, es etwa um eine bestimmte Anzahl von Tagen "in die Zukunft verlegen".
  - Alle möglichen weiteren Operationen könnten gebraucht werden…

Eine Datumsklasse als `struct`

Einführung von Funktionen, die zum Typ passen.

- Erweiterung des Typs durch eigene Methoden (Memberfunktionen) für die gewünschten Operationen.

```
struct Date {
   int y, m, d; // Jahr, Monat, Tag
   void initDate( int yy, int mm, int dd ) { // neues Date
      y = yy, m = mm, d = dd;
   }
   void add_days( int n ){ /*…*/ } // n=1, morgen
};

Date heute{};
heute.initDate( 2011, 11, 21 );
cout << heute.y << "-" << heute.m << "-" << heute.d;
```

- Ausblick: für bestimmte Operationen werden vom Compiler ggf. standardmäßig rudimentäre Methoden automatisch erzeugt, etwa
  - bestimmte Initialisierungen,
  - Zuweisungen,
  - kopieren und verschieben.

Eine Datumsklasse als `struct`

Es wird problematisch...

- Alle Elemente des `struct` sind von überall beliebig zugreifbar.
  - Jahr, Monat und Tag können von überall im Programm beliebig gesetzt werden.
  - Speziell auch beliebig falsch...
- Die Struktur wird also nicht konsequent durchgesetzt.
- Man wünscht sich aber so etwas:

```
Date heute{ 2011, 9, 4 }; // ok

Date weihnachten = Date{ 2011, 12, 24 }; // auch ok

Date pfingsten; // soll Fehler sein: nicht initialisiert

heute.m = 13; // soll Fehler sein

Date irgendwann{ 30, 5, 2037 }; // ?
```

Eine Datumsklasse als `class`

Gekapselte Datenmember und Konstruktoren.

- Jetzt als `class` mit gekapselten Datenmembern.
- Deklaration einer speziellen Memberfunktion, die wir schon bei den Typen unseres Mini-Rechners eingesetzt haben: ein Konstruktor.
  - (Wir hatten uns beim `Token` / `Token_stream` schon etwas genauer mit Konstruktoren auseinandergesetzt.)

```
class Date {
      int y, m, d; // Jahr, Monat, Tag: Default fuer class ist private
                   // nicht mehr "von aussen" zugreifbar.
  public:
      Date( int y, int m, int d ); // Neues Date durch
                                   // Konstruktor.
};
```

Eine Datumsklasse als `class`

Gekapselte Datenmember und Konstruktoren.

- Konstruktoren sind besondere Methoden.
  - Tragen den selben Namen wie ihre Klasse.
  - Geben nichts zurück.
  - Werden zur Initialisierung ("Konstruktion") von Objekten ihrer Klasse verwendet und helfen, den Typ (und damit das gesamte Programm) vor nicht initialisierten Objekten zu schützen.
- Konstruktoren werden mit einer eigenen, speziellen Syntax aufgerufen, die wir schon benutzt haben:

```
Date heute{ 2011, 9, 4 }; // ok

Date weihnachten = Date{ 2011, 12, 24 }; // auch ok

Date pfingsten{}; // Hm?
```

Eine Datumsklasse als `class`

Implementierung `private` halten, Schnittstelle über sorgfältig definierten Methoden zur Verfügung stellen.

- Schutz des Programms vor ungültigen Datumswerten.

```
class Date {
    int y, m, d; // Jahr, Monat, Tag
  public:
    Date( int y, int m, int d ); // Konstruktor
    void add_days( int n );      // aendere um n Tage (z.B. 28.Feb. + 1 Tag)
};
```

- Wir versuchen, Typen so zu entwerfen, dass die Werte ihrer Objekte *garantiert* gültig sind.
  - Solange (Datums-) Werte direkt von überall zugänglich sind wird früher oder später etwas passieren, das einen ungültigen (Datums-) Wert zur Folge hat.
  - Um das zu verhindern sind `private` Member einer Klasse nur über ihre `public` Methoden zugreifbar, die speziell dafür sorgfältig definiert werden müssen: man spricht von *Kapselung*.
- Die Werte eines Objekts werden auch als sein *Zustand* bezeichnet.
- Typen sollen den *gültigen Zustand* ihrer Objekte garantieren.

Eine Datumsklasse

Die Member definieren.

- Die Implementierung der Klasse, also der `private` Teil, wird allgemein üblich ans Ende der Klasse gestellt.

```
class Date {
  public:
    Date( int y, int m, int d );   // Konstruktor
    void add_days( int n );        // n Tage spaeter
  private:
    int  y, m, d;              // gekapselt: Jahr, Monat, Tag
};
```

- Zur Definition eines Members im Quellcode außerhalb seiner Klasse muss man angeben, zu welcher Klasse er gehört.
  - Syntax `klassenname::membername`
  - D.h. man setzt den der Operator `::` ein, um auszudrücken, *welches* `add_days` nun definiert wird.
    - Der Name `add_days` kann in anderen Scopes anders definiert werden.

Eine Datumsklasse

Einen Konstruktor definieren, um nicht initialisierte `Date` Objekte zu verhindern.

- Definition des Konstruktors mittels allgemeiner Syntax für Funktionen.

```
Date::Date( int yy, int mm, int dd ) { // Konstruktor
      y = yy; m = mm; d = dd;
}
```

- Definition des Konstruktors durch spezielle `:` Syntax.
  - Sog. *Memberinitialisierungs-Notation* für Konstruktoren, der bevorzugte Weg.

```
Date::Date( int yy, int mm, int dd )  // Konstruktor in Member-
      : y{ yy }, m{ mm }, d{ dd } { } // initialisierungs-Syntax.
```

- Zur Verdeutlichung das folgende Beispiel:

```
int x = 2;     // Definition und Initialisierung.
int x{2};      // Alternative, verbesserte Notation zu int x = 2;
               // durch die {}-Syntax.
int x(2);      // Alternative Notation, ()-Syntax.

int x; x = 2; // Zuerst nicht initialisierte Definition,
              // dann Zuweisung: nicht zu empfehlen.
```

Eine Datumsklasse

Exkurs: Konstruktoren und andere Methoden direkt innerhalb der Klasse definieren.

- Methoden können auch direkt in der Klassendefinition definiert werden.
  - Sollte im Prinzip vermieden werden, weil dadurch die Klassendefinition schnell unübersichtlich wird.
  - Andererseits kann man sehr kurze und unkomplizierte Funktionen auch ausnahmsweise gleich in der Klasse definieren, dies hat dann zwei Effekte (werden im Rahmen dieser Lehrveranstaltung nicht vertieft):
    - Die Methode wird `inline`, d.h. der Compiler versucht, Objektcode zu erzeugen, der die Funktion ausführt, ohne dass dafür tatsächlich ein Funktionsaufruf erforderlich ist (das ist ein Vorteil für die Performance bei der Ausführung des Programms).
    - Alle Quellcode-Dateien, die die Klasse verwenden, müssen nach Änderungen der `inline` Funktion neu kompiliert werden, was für größere Programme ein Nachteil sein kann (ansonsten ist das nur nötig, wenn die Klassendefinition selbst geändert wurde).
  - Folge: Methoden nicht innerhalb der Klasse definieren, außer es handelt sich um sehr kleine Methoden mit hohen Performance-Anforderungen.

Eine Datumsklasse

Invarianten und Fehlerbehandlung.

- Was soll bei einem ungültigen Datum passieren?
  - Sie kennen die Antwort: das Programm soll eine Ausnahme auslösen.
  - Zunächst muss bei der Erzeugung eines `Date` Objekts dafür gesorgt werden, dass keine `Date` Objekte mit ungültigem Zustand entstehen können.

```
class Date {
 public:
   class Invalid {}; // Typ fuer Objekte, die (nur) einen Date Fehler anzeigen.
   Date( int y, int m, int d ); // Konstruktor-Deklaration.
    // …
 private:
   int  y, m, d; // Jahr, Monat, Tag.
   bool check( ); // private Methode, gibt true nur bei gueltigem Datum.
};

Date::Date( int yy, int mm, int dd ) // Konstruktor-Definition.
   : y{ yy }, m{ mm }, d{ dd }
{ if( !check() ) throw Invalid{}; }
```

- Die Regeln, die gültige (Datums-) Werte definieren, werden Invarianten genannt.
  - Die Prüfung wird in eine separate Funktion verlagert.

```
bool Date::check( ) {
   if( /* Invarianten verletzt */ ) return false; return true; }
```

Eine Datumsklasse

Die `bool Date::check()` Methode, die Datumsangaben auf ihre Gültigkeit prüft.

- Eine "industriefähige" Lösung sprengt hier bei Weitem den Rahmen.
  - Wir wollen uns zur Vertiefung trotzdem ein wenig mit der Frage beschäftigen, welche kritischen Fehlerquellen es gibt.
- Fehlerquelle Jahreszahl
  - Gregorianischer Kalender: 10. Okt. 1582...
  - Wir erlauben nur Jahreszahlen zwischen 1800 und 2200.
- Fehlerquelle Tag im Monat
  - 31. April...
  - Wir prüfen und beachten den Februar besonders (Schaltjahresregel).
- Fehlerquelle Monat und Reihenfolge der Parameter
  - `2011,12,3` als 12. Mrz. 2011 oder 3. Dez. 2011 ist kulturabhängig.
  - Wir lassen für Monate keine Zahleneingaben sondern nur Abkürzungen wie *Mrz* oder *Dez* zu (Implementierung: als Aufzählung mit einem Enumerator pro Monat, innerhalb der `Date` Klasse).

Eine Datumsklasse

Die `bool Date::check()` Methode, die Datumsangaben auf ihre Gültigkeit prüft.

- Wichtige Teile der `Date` Klasse bis hier:

```
class Date {
  public:
     class Invalid {}; // Ausnahmen anzeigen
     enum Month { Jan=1, Feb, Mrz, Apr, Mai, Jun,   // sym. Konst.
                  Jul, Aug, Sep, Okt, Nov, Dez };   // fuer Monate
     Date( int y, Month m, int d ); // ein Konstruktor
     // …
  private:
     int y;
     Month m;
     int d;
     bool check();
};
```

- Definition der `Date::check()` Methode, die die Invarianten prüft?

Eine Datumsklasse

Die `bool Date::check()` Methode, die Datumsangaben auf ihre Gültigkeit prüft.

- Eine mögliche Definition der `Date::check()` Methode:

```
bool Date::check() {
    // Jahr einschraenken:
   if( y < 1800 || 2200 < y ) return false;
   // falsche Monate verhindern
   // was passieren kann, z.B. durch Date::Month(13)
   if( m < Jan || Dez < m ) return false;
   // Tageswerte unter 1 verhindern:
   if( d < 1 ) return false;
   // zu grosse Tageswerte verhindern:
   switch( m ) {
      case Feb: // noch ohne Schaltjahre
         if( 28 < d ) return false;
      case Apr: case Jun: case Sep: case Nov:
         if( 30 < d ) return false;
      default:
         if( 31 < d ) return false;
     }
  return true;
}
```

Eine Datumsklasse

Der Standardkonstruktor (default constructor) ist derjenige Konstruktor, der keine Parameter erhält.

- Durch den Standardkonstruktor einer Klasse wird sichergestellt, dass jedes Objekt der Klasse immer komplett initialisiert ist.
  - Damit man nicht mehr vergessen kann, für ein Objekt der Klasse initiale Werte zu setzen.
- Für viele Typen gibt es sinnvolle Standardwerte, die eine natürliche Initialisierung per Standardkonstruktor auch ohne ausdrückliche Angabe von Anfangswerten möglich machen.

```
string s1{};     // Standard: die leere Zeichenkette ""
vector<T> v1{}; // Standard: der leere vector, null Elemente vom Typ T
```

- Für einen Typ `T`
  - ist **`T()`** die Notation des Standardwerts, wie durch den Standardkonstruktor bestimmt,
  - ist **`T{}`** die `{}`-Syntax zur Initialisierung (Wertkonstruktion), z.B. mittels Standardkonstruktor (stärker typgeprüft als die `()`-Syntax).

```
string s1{}; // alternative Notation: string s1; oder string s1();
vector<string> v1 = vector<string>{}; // alternativ: analog wie oben
vector<string> v2( 10, string{} ); // vector mit 10 leeren Zeichenketten
string{}; string(); // andere Schreibweisen fuer ""
double{}; double(); // andere Schreibweisen fuer 0.0
```

Eine Datumsklasse

Standardkonstruktor für `Date` Objekte.

- Für `Date` Objekte muss der Programmierer beim Aufruf des bereits von uns definierten Konstruktors drei Parameter übergeben: Jahr, Monat, und Tag.
- Da es für Datumsangaben keinen völlig offensichtlichen Standardwert gibt, könnte man für den Standardkonstruktor etwa 1. Januar 1900 nehmen.

```
class Date {
  public:
     Date( );            // Deklaration Standard-Konstruktor
     // …                //   heisst wie die Klasse (wie alle Konstruktoren)
                         //   hat zusaetzlich eine leere Parameterliste
  private:
     int y, d;
     Month m;
     // …
};
Date::Date( )                  // Definition Standard-Konstruktor
  : y{1900}, m{Jan}, d{1}      // Memberinitialisierungs-Notation
{ }                            // leerer Methodenrumpf

// alternativ: Date::Date( ) : Date{ 1900, Jan, 1 } { }
// sog. delegierender Konstruktor.
// Der andere Konstruktor, der dabei aufgerufen wird (das
// "Delegationsziel"), muss natuerlich existieren.
```

Eine Datumsklasse

Vertiefung zum Standardkonstruktor für `Date` Objekte.

- Wir wollen zur Vertiefung noch eine Supportfunktion definieren, die dem `Date` Standardkonstruktor einen Standardwert liefert.

```
const Date& default_date( ) {
   static const Date dd{ 1900, Date::Jan, 1 };
   return dd;
}
```

  - Diese Funktion erzeugt ein `Date` Objekt namens `dd` (als `const` qualifiziert, damit die aufrufende Funktion es nicht versehentlich ändert) und gibt eine Referenz auf `dd` zurück, um unnötiges Kopieren zu vermeiden.
  - `dd` ist außerdem als `static` qualifiziert: lokale Variablen, die als `static` qualifiziert sind, werden einmalig (nur) beim ersten Aufruf der Funktion erzeugt.
  - Beachten Sie auch die Syntax `Date::Jan` (*nicht* `Date.Jan`).
    - `Date` ist kein Objekt sondern eine Klasse.
    - Verwenden Sie `::` nach Klassen und Namensbereichen und `.` nach Objekten.
- *Aufgabe:* wie sieht ein Standardkonstruktor dazu aus?

Eine Datumsklasse

Vertiefung zum Standardkonstruktor für `Date` Objekte.

- Delegierender Standardkonstruktor mit `default_date()` Aufruf:

```
Date::Date() :
  Date{ default_date().y, default_date().m, default_date().d } { }
```

- Mit einem Standardkonstruktor für den `Date` Typ kann man jetzt z.B. folgende Anweisung schreiben:

```
vector<Date> geburtstage( 10 );
```

- Ohne Standardkonstruktor müsste man z.B. schreiben:

```
vector<Date> geburtstage( 10, default_date() );
```

- Bemerkungen:
  - Das Standarddatum soll natürlich *nicht* als globale Variable angelegt werden.
  - Die Supportfunktion `default_date()` verhält sich mit der `static` Variablen jedoch ähnlich wie eine globale Variable und kann daher eine geeignete Lösung sein.

Eine Datumsklasse

Automatisch generierte Methoden.

- Für bestimmte Operationen – und unter bestimmten Umständen – werden vom Compiler standardmäßig rudimentäre Methoden automatisch erzeugt:
  - Standardkonstruktor,
  - Zuweisungen,
  - kopieren und verschieben,
  - sog. Destruktor.
- Wir werden in L3 noch genauer darauf zurückkommen,
  - welche Funktionalität die noch nicht näher angesprochenen und ggf. automatisch generierten Methoden implementieren
  - und unter welchen Umstände die Methoden generiert werden oder nicht.

Eine Datumsklasse

Konstante Methoden.

- Die Operationen einer Klasse (Methoden) können und sollten klassifiziert werden als
  - verändernd (ggf. schreibend),
  - nicht verändernd (rein lesend).
- Um anzuzeigen, dass eine Methode nur lesend auf die Datenmember des Objekts zugreift und damit auch für `const` Objekte aufgerufen werden kann, deklariert man `const` direkt hinter der Parameterliste der Methode.
- Der Compiler überwacht dann, dass nur `const` Operationen, die das Objekt nicht ändern, für `const` Objekte aufgerufen werden (und übersetzt den Quellcode andernfalls nicht).
- Beispiele finden Sie im vorlesungsbegleitenden Quellcode zur `Date` Klasse.

Eine Datumsklasse

Methoden, die vielleicht besser keine sind: sog. Support-Funktionen.

- Eine Funktion, die problemlos und elegant außerhalb der Klasse implementiert werden kann, sollte eine eigenständige Funktion und keine Methode der Klasse sein.
- Viele dieser Funktionen (nicht alle) nehmen Aufrufparameter der Klasse entgegen, die sie unterstützen.
- Oft werden Namensbereiche verwendet, um zusammen gehörende Support-Funktionen zu identifizieren.
- Hauptgrund: die Klasse bleibt dann kleiner und damit besser wartbar.
  - Beispielsweise haben kommerziell angebotene Datumsbibliotheken, die meist rein auf Bequemlichkeit für die benutzenden Programmierer (anstelle von Verständlichkeit und Wartbarkeit) ausgelegt sind, schnell 50 und mehr Schnittstellen-Methoden.
  - In solchen Klassen lassen sich grundlegende Dinge selten noch ändern (z.B. falls man das Datum durch die Anzahl der Tage seit dem 1. Jan. 1800 darstellen wollte).
- Klassisches Beispiel für eine Support-Funktion beim Datumstyp wäre:

```
bool leap_year( int year ); // true falls Schaltjahr
```

Nochmal zur Initialisierung...

Für die Initialisierung stehen unterschiedliche Syntaxformen zur Verfügung.

- **`T t1 { v };`**
  - Sog. Listen-Initialisierung: *nur* diese Syntax kann in jedem Zusammenhang eingesetzt werden, der Stil ist bevorzugt zu verwenden (Einschränkung: `auto`).
  - Regeln bei Mehrdeutigkeit, z.B.:
    `struct T { T( initializer_list<int> ); T( int ); T( ); int i; };`
    - Gibt es den Standardkonstruktor und den Listenkonstruktor, wird der Standardkonstruktor aufgerufen:
      `T x {}; // Standardkonstruktor`
    - Gibt es den Listenkonstruktor und einen passenden "normalen" Konstruktor ("nicht-Standardkonstruktor"), wird der Listenkonstruktor aufgerufen:
      `T x {1}; // Listenkonstruktor`
- **`T t2 = { v };`** `// mischt "C-Stil" mit C++`
- **`T t3 = v;`** `// reiner "C-Stil"`
- **`T t4( v );`** `// direkter Konstruktoraufruf`

Ausblick:
Wir werden noch näher drauf eingehen, wie wir die oben erwähnten `initializer_list<S>` Konstruktoren selbst definieren können...

LEERE SEITE

Operatorenüberladung

Was ist Operatorenüberladung (operator overloading)?

- Die Bedeutung von Operatoren kann für deren Einsatz mit Operanden, die Objekte eines benutzerdefinierten Typs sind, spezifisch definiert werden.
  - Die Technik wird als Operatorenüberladung bezeichnet.
  - Damit kann einem benutzerdefinierten Typ eine eher konventionell anmutende Notation für seine Verwendung verpasst werden.
    - Es können zwar keine eigenen Operatorsymbole definiert werden, man kann aber fast alle der in C++ bekannten Operatorsymbole umdefinieren.
    - Nicht überladen kann man z.B.:
      - den Punktoperator `.`
      - den Scope Resolution Operator `::`
      - den ternären Operator `?:`
- Man kann einen Operator nur mit der üblichen Anzahl von Operanden, der üblichen Priorität und der üblichen Assoziativität umdefinieren.
- Man kann einen Operator natürlich nur überladen, wenn er mindestens einen benutzerdefinierten Typ als Operanden hat.
- Zum Überladen schreibt man einfach eine eigene, entsprechende Operatorfunktion.

Operatorenüberladung

Operatoren und Operatorfunktionen als unterschiedliche Schreibweisen mit der selben Bedeutung.

- Die Operator-Syntax ist (nur) eine bequeme Kurzschreibweise für den Aufruf der entsprechenden Operator-Funktion.

```
string s1{ "hallo, " };   // Initialisierung
cout << s1;               // Operator << (stream insertion)
s1 += "operator";         // Operator +=
cout << s1 << "!\n";      // Operator << (stream insertion)

string s2{ "hallo, " };
operator<<( cout, s2 );
s2.operator+=( "operatorfunktion" );
operator<<( operator<<( cout, s2 ), "!\n" );
```

- Praxistipp: überladen sie Operatoren nur, wenn Sie wirklich gute Gründe dafür haben.
  - Überladene Operatoren können kräftig für Verwirrung sorgen (Operatoren-überladung gilt in der Praxis als eher verwirrendes Sprachmittel).
  - Ausnahme: es würde den Quellcode entscheidend verbessern, was zwar selten ist, aber vorkommt, z.B. bei den (noch zu besprechenden) Iteratoren oder bei der Stromausgabe.

Operatorenüberladung

Binäre und unäre Operatorfunktionen.

- Überladene binäre Operatoren `xx ■ yy`
  - Entweder als Methoden mit einem Parameter: `xx.operator ■ ( yy )`
  - Oder als globale Supportfunktionen mit zwei Parametern: `operator ■ ( xx,yy )`
- Überladene unäre Präfix-Operatoren `■ zz`
  - Entweder als Methoden ohne Parameter : `zz.operator ■ ( )`
  - Oder als globale Supportfunktionen mit einem Parameter: `operator ■ ( zz )`
- Überladene unäre Postfix-Operatoren `zz ■`
  - Entweder als Methoden mit einem Parameter : `zz.operator ■ ( int )`
  - Oder als globale Supportfunktionen mit zwei Parametern: `operator ■ ( zz,int )`
  - Der Wert im `int`- Parameter wird nie benutzt, der `int`- Parameter selbst ist ein reiner Dummy-Parameter zur Unterscheidung zwischen Präfix- und Postfix-Version: der vorhandene Parameter dient zur Kennzeichnung der Postfix-Version.

Operatorenüberladung

Zur Vertiefung: ein überladener Operator zur Ausgabe für `Date` Objekte (i) mit sog. get-Methoden ("Gettern") an der Schnittstelle.

- Überladener Operator << (stream insertion) zur Ausgabe von `Date` Objekten (ii).

```
class Date {
  public:
    // …
    int get_y( ) const { return y; }
    int get_m( ) const { return m; }
    int get_d( ) const { return d; }
  private:
    //…
};
// Operator definieren
ostream& operator<<( ostream& os, const Date& d ) {
  return os << '('
            << d.get_y() << '-'
            << d.get_m() << '-'
            << d.get_d() << ')';
}
```

- Die hier gezeigte Bereitstellung entsprechender Methoden an der Schnittstelle ist eine oft verwendete Technik.
- Im begleitenden Quellcode zur `Date` Klasse finden Sie noch ein weiteres Beispiel für Operatorenüberladung.

Operatorenüberladung

Zur Vertiefung: ein überladener Operator zur Ausgabe für `Date` Objekte (ii) als `friend`.

- Überladener Operator << (stream insertion) zur Ausgabe von `Date` Objekten (i).

```
class Date {
  public:
     //…
  private:
     //…
     int  y, m, d; // Jahr, Monat, Tag
     friend ostream& operator<<( ostream&, const Date& );
};
// Operator definieren
ostream& operator<<( ostream& os, const Date& d ) {
  return os << '(' << d.y << ',' << d.m << ',' << d.d << ')';
}
```

- Operatoren werden ähnlich wie Funktionen definiert.
  - Syntax: `RT operator■( TA a, TB b );`
- `friend` Member in der Klasse (hier: die Operatorfunktion) gehören nicht zur Klasse, haben aber dennoch auf Klassenmember Zugriff.
  - Das kann manchmal erforderlich sein und stellt *keine* Verletzung der Kapselung dar. (Frage: warum nicht?)

Exkurs

Rundung.

- Für eine reelle Zahl r definieren wir die unären Operatoren *Abrundungsklammer* und *Aufrundungsklammer*:
  - $\lfloor r \rfloor$ ist die größte ganze Zahl kleiner oder gleich r
    - Funktion der C++ StdLib zur Abrundung: **`floor()`**
  - $\lceil r \rceil$ ist die kleinste ganze Zahl größer oder gleich r
    - Funktion der C++ StdLib zur Aufrundung: **`ceil()`**
- Bemerkungen
  - $\lceil r \rceil = \lfloor r \rfloor \quad \Leftrightarrow r$ ist eine ganze Zahl
  - $\lceil r \rceil = \lfloor r \rfloor + 1 \Leftrightarrow r$ ist keine ganze Zahl
  - $\lfloor -r \rfloor = -\lceil r \rceil; \quad \lceil -r \rceil = -\lfloor r \rfloor;$
  - $r - 1 < \lfloor r \rfloor \leq r \leq \lceil r \rceil < r + 1$
- Aufgabe: schreiben Sie mittels `std::floor()` und `std::ceil()` eine `myround()` Funktion, die sich wie `std::round()` verhält,
  - die also ab ",5" aufrundet und ansonsten abrundet.

Exkurs

Modul (Teilungsrest).

- Für zwei reelle Zahlen x und y kann man eine binäre Operation mod (Modul bzw. Teilungsrest) spezifizieren:
  - $x \bmod y = x - y \lfloor x/y \rfloor$, mit $y \neq 0$ und $x \bmod 0 = x$
  - Also der Rest bei der Division von x durch y.
  - Vorsicht: der Moduloperator in Programmiersprachen (in C++ der Operator `%`) verhält sich nicht unbedingt genau so.
- Bemerkungen zu dieser Spezifikation:
  - $0 \leq \left(\frac{x}{y}\right) - \left\lfloor \frac{x}{y} \right\rfloor = \frac{x \bmod y}{y} < 1$
  - $y > 0 \Rightarrow 0 \leq x \bmod y < y$
  - $y < 0 \Rightarrow 0 \geq x \bmod y > y$
  - $x - (x \bmod y)$ ist ein ganzzahliges Vielfaches von y
- Aufgabe: schreiben Sie eine `modulo()` Funktion nach der obigen Spezifikation.

Exkurs

Kongruenz.

- $x \equiv r \pmod{y}$
  - Aussprache: "x ist kongruent zu r, modulo y".
  - Bedeutung: r ist der Rest bei der Division von x durch y.
    - D.h. $(x - r)$ ist ein ganzzahliges Vielfaches von y.
    - $x \equiv r \pmod{y}$ bedeutet auch $x \bmod y = r \bmod y$.
- Bemerkungen
  - $a \equiv b,\ x \equiv y \ \Rightarrow\ a \pm x \equiv b \pm y,\ ax \equiv by \pmod{m}$
  - $a \equiv b \pmod{m} \ \Leftrightarrow\ an \equiv bn \pmod{mn},\ n \neq 0$

Einige Beispielfragen

Klassen.

- Welches sind die beiden Teile einer Klasse, die Sie bereits kennen gelernt haben?
- Erklären Sie den Unterschied zwischen der Schnittstelle und der Implementierung einer Klasse.
- Welche Probleme entstehen bei der Behandlung von (Datums-)Werten als `struct`?
- Was ist der entscheidende Unterschied zwischen `struct` und `class`?
- Warum wird für die Klasse der Datumswerte ein Konstruktor verwendet, und nicht einfach eine Funktion oder Methode wie `initDate()`?
- Was ist eine Invariante? Geben Sie fünf *eigene* Beispiele.
- Wann würden Sie eine Funktion als Methode einer Klasse definieren, wann würden Sie sie außerhalb der Klasse definieren? Warum?
- Warum sollte man in einem Programm im Allgemeinen Operatoren nur dann überladen, wenn es wirklich gute Gründe dafür gibt?

Einige Beispielfragen

Klassen.

- Warum sollte die Schnittstelle einer Klasse so klein wie möglich gehalten werden?
- Welche Wirkung hat `const` in dieser Deklaration:
  `void Date::print() const;`
- Warum sollten Support-Funktionen keine Methoden sein?
- Welche Syntax ermöglicht den Zugriff auf Objektmember? Können Sie mit dieser Syntax immer auf die Member zugreifen, oder gibt es Einschränkungen (ggf. welche)?
- Welche Syntax ermöglicht die Definition von Methoden im Quellcode außerhalb der Klasse?
- Erklären Sie den entscheidenden Unterschied zwischen einem Konstruktor und dem Standardkonstruktor.
- Welche vier Formen der Initialisierung gibt es? Welche ist zu bevorzugen? Warum?
- Erläutern Sie den Nutzen des Standardkonstruktors an einem *eigenen* Beispiel.

Nächste Einheit:
Einfaches Suchen und Sortieren