§7 Einfache Klassen - Allgemeines

Leitideen: *Ausgangspunkt für Klassen in C++ sind C-Records (struct), die mit speziellen Funktionen und Zugriffsattributen angereichert werden.*

Die Analyse der allgemeinen Eigenschaften von Datentypen legt einen Mechanismus zur Realisierung benutzerdefinierter Datentypen (Klassen) nahe.

Wesentliche Eigenschaften von Datentypen sind die Beschreibung zulässiger Operationen auf den Werten und die Trennung von Benutzerschnittstelle und Implementierung.

Operatoren für Klassen werden durch Operatorfunktionen bereitgestellt. (Eingebaute Datentypen sind allerdings nicht veränderbar.)

Die wiederholte Spezifikation strukturell gleicher Klassen kann durch Templates (datentypabhängige Klassen und Funktionen) vermieden werden.

§7 Einfache Klassen - Themenübersicht

- Records
- Benutzerdefinierte Datentypen
- Klassenkonzept in C++
- Beispiel - Komplexe Zahlen
- Andere interne Darstellung komplexer Zahlen
- Begriffsübersicht
- Zugriffsattribute
- Funktionen innerhalb von Klassen
- Überladen von Operatoren I,II
- Überladen von Funktionen I,II
- Templates
- Komplexe Zahlen in der Standardbibliothek
- Initialisierung durch und Zuweisung von Klassenobjekten
- Ressourcenallokation und -deallokation

Records

Vektor Zusammengesetzter Datentyp aus Komponenten *gleichen* Typs, Komp.zugriff über Index (z.B. `a[3]`)

Record Zusammengesetzter Datentyp aus Komponenten evtl. *unterschiedl.* Typs, Komp.zugriff über Name (z.B. `c.x`)

- Bsp. für typische Vereinbarung:
  `struct Complex{...};` *meist außerhalb von Fkt.*
  :
  `Complex z,w;` *z,w Recordvariablen*
- Komponentenzugriff: `c.x`, `cp->x` $\hat{=}$ `(*cp).x`
- Im Unterschied zu Vektoren für Records möglich: Test auf Gleichheit, Zuweisung, Parameterübergabe und Rückgabe als Funktionswert
- In C als Komponenten lediglich einfache Datentypen, C-Vektoren, Zeiger, Records bzw. Kombinationen erlaubt
- In C auch keine Zugriffsattribute und Funktionen
- In C++ `struct` und `class` austauschbar, aber unterschiedliche Voreinstellung für Zugriffsattribute

Benutzerdefinierte Datentypen

- Datentypen bestehen aus einer Menge von Werten und darauf zulässigen Operationen. (Bsp.: int, double etc.)
- Bei Datentypen gibt es eine Trennung von Benutzerschnittstelle (Programmierschnittstelle) und Implementierung (Bitdarstellung und Interpretation der Werte etc.)
- Realisierung prinzipiell möglich durch datentypspezifische Modifikation des Compilerquellcodes, jedoch völlig impraktikabel.
- Operatoren für Datentypen können mit Hilfe von Operatorfunktionen ausgedrückt werden (z.B. `a+b`→`add(a,b)`), diese Umsetzung kann beim Übersetzen des Programms vorgenommen werden.
- Somit reduziert sich die Definition von Datentypen auf die Festlegung der Speicherbereiche für die Werte und die Definition von Funktionen – am besten in enger räumlicher Nähe. Insbesondere wird eine Funktion benötigt, die bei der Variablendefinition automatisch aufgerufen wird und den Speicherplatz passend initialisiert (Konstruktor).

Klassenkonzept in C++

Zweck

- Realisierung benutzerdefinierter Datentypen durch Spezifikation eines Records für die Daten und Definition geeigneter Funktionen bzw. Operatorfunktionen, die darauf operieren.
- Einschränkung des Zugriff auf die Klassenkomponenten (Daten und Funktionen) – dieser ist nur bestimmten Funktionen gestattet. Dadurch: Schaffung einer Schnittstelle nach außen.

Unterschiede zu C-Records (Auszug)

- Innerhalb von Klassen können auch Funktionen auftreten:
  Konstruktoren – Initialisierung von Objekten (z.B. Variab.)
  Destruktoren – Freigabe von Objekten
  weitere Komponentenfunktionen – Aufruf: c.f() (für Var. c)
  friend-Funktionen – Aufruf: f(c)
- Zugriffsattribute:
  `private` – Zugriff nur für Komp.fkt. und friend-Fkt.
  `public` – Zugriff für beliebige Funktionen

Beispiel - Komplexe Zahlen

```
class Complex {
  private:
    double re,im;  // Datenkomponenten
  public:
    // Konstruktor mit Initialisierungsliste
    Complex(double Re=0,double Im=0):
      re(Re),im(Im) {}
    double real() {return re;} // Komponentenfkt.
    double imag() {return im;} // Komponentenfkt.
  };

int main()
{
  Complex null, eins(1.0), i(0.0,1.0);
  cout << "Re i = " << i.real() << endl;
  cout << "Im i = " << i.imag() << endl;
}
```

Beispiel - Komplexe Zahlen II

Bedeutung der Konstruktordefinition

- Der Konstruktor ist hier eine Funktion, die die Datenkomponenten `re` und `im` mit Werten belegt
- Konstruktorname = Klassenname
- *Kein* Ergebnistyp, auch *nicht* `void`
- Definiert werden genau genommen 3 Konstruktoren (mit 0, 1 bzw. 2 Parameter):

```
Complex(): re(0),im(0) {}
Complex(double Re): re(Re),im(0) {}
Complex(double Re,double Im): re(Re),im(Im) {}
```

- Verzicht auf Konstruktorinitialisierungsliste hier möglich:

```
Complex() {re=0; im=0;}
Complex(double Re) {re=Re; im=0;}
Complex(double Re,double Im) {re=Re; im=Im;}
```

- Letzteres zusammengefasst:

```
Complex(double Re=0,double Im=0) {re=Re; im=Im;}
```

Bsp. - Komplexe Zahlen (andere interne Darstellung)

Ziel: $x + iy = re^{i\varphi}$ nicht als (x, y) speichern, sondern als (r, φ)

```
class Complex {
  private:
    double r,phi;  // Datenkomponenten
  public:
    // Konstruktor mit Initialisierungsliste
    Complex(double Re=0,double Im=0):
       r(sqrt(Re*Re+Im*Im)), phi(atan2(Im,Re)) {}
    double real() {return r*cos(phi);}
    double imag() {return r*sin(phi);}
  };

int main()
{
  Complex null, eins(1.0), i(0.0,1.0);
  cout << "Re i = " << i.real() << endl;
  cout << "Im i = " << i.imag() << endl;
}
```

Begriffsübersicht (vereinfacht)

Klasse	Datentyp, gebildet aus Recordkomponenten + Komponentenfunktionen + Zugriffsattribute
Objekt	Ausprägung einer Klasse, z.B Variable vom Datentyp der Klasse „Objekt ist Instanz einer Klasse“
Konstruktor	Komponentenfunktion zur Erzeugung bzw. Initialisierung von Objekten
Destruktor	Komponentenfkt. zur Freigabe von Objekten
Komponentenfkt. (Elementfkt., Memberfkt., Methode)	Klassenkomponente vom Funktionstyp hat Zugriff auf alle Klassenkomponenten Aufruf außerhalb der Klassenvereinb.: `c.f()` (Ausnahme: Konstruktor/Destruktor)
befreundete Fkt.	Funktion mit Zugriff auf alle Klassenkomp. Aufruf außerhalb der Klassenvereinb.: `f()`
Zugriffsattribut	Kennzeichnung einer Klassenkomponente, die angibt, welche Funktionen Zugriff auf sie haben.

Zugriffsattribute

Klassenkomponenten können mit einem Zugriffsattribut versehen werden.

Attribut	*Bedeutung*	*Voreinstellung für*
`private`	Nur Komponentenfunktionen und befreundete Funktionen haben Zugriff auf die Komponente	`class`
`public`	Alle Funktionen haben Zugriff auf die Komponente	`struct`

- `struct` wird nur selten in C++-Programmen benutzt.

Funktionen innerhalb von Klassen (Auszug)

Die Syntax entscheidet über die Bedeutung einer Funktion `f`, die innerhalb einer Klasse C definiert ist.

Bsp.:
```
class C {...};   // Definition der Klasse C
C c;             // c hat Datentyp C
```

Funktionskopf	*Bedeutung*	*Aufruf außerhalb der Klasse*
`C(...)`	Konstruktor	`C c(...) //indirekt` `c=C(...) //direkt`
`~C()`	Destruktor	`~C() //meist implizit`
`T f(...)`	Komponentenfkt.	`c.f(...)`
`friend T f(...)`	befreundete Fkt.	`f(...)`

- Konstruktoren werden *ohne* Ergebnistyp vereinbart.
- Bestimmte Konstruktoren werden automatisch erzeugt, falls sie nicht definiert sind.
- Initialisierung von Datenkomponenten erfolgt häufig über die Konstruktorinitialisierungliste.
- Komponentenfkt. können direkt die Komponentennamen verwenden. Diese beziehen sich auf die Komp. von `c`.

Überladen von Operatoren I (Auszug)

Prinzip: x`@`$y \rightarrow$ `operator@(`x, y`)`

- `operator@`: benutzerdef. Fkt. oder Bibl.fkt.
- Def. der Operatorfkt. auch außerhalb der Klasse möglich
- Verschiedene Funktionen `operator@` zulässig, Auswahl nach Parametertypen („Überladen des Funktionsnamens“)
- Überladen von unären Präfixoperatoren durch Operatorfunktionen mit einem Parameter
- Operatorenvorrang und Syntax wie bei den eingebauten Datentypen

Beispiele:

1. Addition komplexer Zahlen

```
Complex operator+(Complex z1, Complex z2)
z1 + z2 → operator+(z1,z2)
```

Überladen von Operatoren II (Auszug)

2. Subtraktion komplexer Zahlen

   `Complex operator-(Complex z1, Complex z2)`

   `z1 - z2` → `operator-(z1,z2)`

3. Negativbildung einer komplexen Zahl

   `Complex operator-(Complex z)`

   `-z` → `operator-(z)`

4. Ausgabe komplexer Zahlen

   `ostream& operator<<(ostream& stream, Complex z)`

   `cout << z` → `operator<<(cout,z)`

   `cout << z1 << z2` → $\underbrace{\texttt{(cout << z1)}}_{\texttt{cout}}$ `<< z2`

5. Eingabe komplexer Zahlen

   `istream& operator>>(istream& stream, Complex& z)`

   `cin >> z` → `operator>>(cin,z)`

   `cin >> z1 >> z2` → $\underbrace{\texttt{(cin >> z1)}}_{\texttt{cin}}$ `>> z2`

   Unterschied zur Ausgabe: `Complex&` **statt** `Complex`

Überladen von Funktionen

- Gleichnamige Funktionen: Ausgewählt wird diejenige, bei der die Argumente am besten zu den Parametertypen passen.
- Entscheidend: Parametertypen, nicht Ergebnistyp.

Definition komplexe Wurzel

$z = re^{i\varphi} \quad (r \geq 0, \varphi \in [-\pi, \pi])$

Allerdings: $e^{i\pi} = e^{-i\pi}$

Funktionentheorie: $\sqrt{z} = \sqrt{r}e^{i\frac{\varphi}{2}} \quad (-\pi < \varphi < \pi)$
(Hauptzweig) d.h. Weglassen der negativen reellen Achse

Hauptwert: $\sqrt{z} = \sqrt{r}e^{i\frac{\varphi}{2}} \quad (-\pi < \varphi \leq \pi)$
insb. $\sqrt{x} = i\sqrt{|x|} \quad (x < 0)$

C 99 (IEC 60559) $\operatorname{Re} z < 0$, $\operatorname{Im} z = \pm 0$: $\varphi = \pm\pi$
$\varphi = \texttt{atan2}(\operatorname{Im} z, \operatorname{Re} z)$
$\sqrt{z} := \sqrt{r}e^{i\frac{\varphi}{2}}$
insb. $\sqrt{z} := \pm i\sqrt{|z|}$, falls $\operatorname{Re} z < 0$ und $\operatorname{Im} z = \pm 0$

Überladen von Funktionen II

Beispiel reelle und komplexe Wurzel

Funktionsköpfe

```
double sqrt(double x)
Complex sqrt(Complex z)
```

Beispiele

`sqrt(2.0)`	$\rightarrow$	$\sqrt{2}$
`sqrt(-2.0)`	$\rightarrow$	`NaN`
`sqrt(Complex(2.0))`	$\rightarrow$	$\sqrt{2}$
`sqrt(Complex(-2.0))`	$\rightarrow$	$\sqrt{2} \cdot i$
`sqrt(-Complex(2.0))`	$\rightarrow$	$-\sqrt{2} \cdot i$

denn:

`Complex(-2.0)`	$\rightarrow$	`(-2.0,+0.0)`	$\rightarrow$	$\varphi = \pi$
`-Complex(2.0)`	$\rightarrow$	`-(2.0,+0.0)`	$\rightarrow$	`(-2.0,-0.0)`
			$\rightarrow$	$\varphi = -\pi$

Templates

Problem: 3 Gleitpunktdatentypen: `float`, `double`, `long double`
Also auch 3 Datentypen für komplexe Zahlen?
Z.B.: `Complex_float`, `Complex_double`, `Complex_long_double`

Lösung: Ja, aber nur eine Definition!
Daher: Klasse mit Typparameter (Klassentemplate)

Aktion	*Klasse*	*Klassentemplate*
Definition	`class C`	`template <class T> class C`
Variablenvereinb.	`C c`	`C<T> c`

Erweiterungen

- Mehrere Typparameter
- Voreinstellungen für Typparameter

Komplexe Zahlen in der Standardbibliothek

- Headerdatei: `complex`
- Datentypen: `complex<float>`, `complex<double>`, `complex<long double>`
- Übliche arithmetische Operatoren: `+ - * /`
- Ein/Ausgabe mit überladenen Shiftoperatoren
  Eingabeformat: `x (x) (x,y)` *Nicht:* `x+i*y` o.ä.
  Ausgabeformat: `(x,y)`
- Real- und Imaginärteil sowohl über normale Funktion als auch Komponentenfunktion verfügbar
- C++-Standard: Komplexe Funktionen über Hauptwerte def.
  Implementierungen z.T. entsprechend IEC 60559,
  d.h. $f(x \pm i \cdot 0) := \lim_{\varepsilon \to 0} f(x \pm i\varepsilon)$

Caveat

- `norm(z)` $= |z|^2$ (!!)
- Keine automat. Umwandlung `int`$\rightarrow$`complex<double>` bei überlad. Operatoren, z.B. in `1+i` mit `i` imag. Einheit

Initialisierung durch und Zuweisung von Klassenobj.

Voreinstellungen

- Initialisierung und Zuweisung erfolgen komponentenweise (Initialisierung durch Klassenobjekte maßgeblich für Wertparameterübergabe und Rückgabe von Funktionswerten)
- Entsprechende Kopierkonstruktoren und Zuweisungsoperatoren werden automatisch erzeugt (sofern nicht definiert)
- Komponente vom Klassentyp: Verwendung der entsprechenden Kopierkonstruktoren und Zuweisungsoperatoren
- Komponente mit skalarem Datentyp: Initialisierung und Zuweisung verhalten sich im wesentlich gleich (abgesehen von const-Attributen)

Beispiele: Komplexe Zahlen, Polynomklasse (Inf.bl.14,15)

Allerdings oft nicht ausreichend! (→ `class Vektor` *– Prog.II)*

Ressourcenallokation und -deallokation

- Ziel von Klassen: Vollständig initialisierte Objekte
- Zuständig: Konstruktor(en)
- Größenveränderliche Objekte werden in der Regel auf dem Heap angelegt
- Konstruktor alloziert Speicherplatz, Destruktor deallloziert zur Ressourcenschonung *und* zur Vermeidung von „Speicherlecks“
- Ähnliche Vorgehensweise auch beim Dateizugriff
- Konzept bekannt unter “Ressource allocation is initialization“ (RAII)

Beispiele: STL-Behältertypen (z.B. `vector<T>`)
Dateistromklassen, Stringstromklassen