Karlsruher Institut für Technologie

C++ Workshop

3. Block, 11.05.2012

Sven Brauch, Robert Schneider | 17. Mai 2012

Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

Gliederung

Karlsruher Institut für Technologie

1. Intro
2. Vererbung
   - Grundlegendes zur Vererbung
   - Virtuelle Methoden
   - Abstrakte Klassen
   - dynamic_cast
3. Das Bitmap-Framework
   - Bitmap24 und Algorithmen
   - BatchBitmap24 und Zeichen-Objekte
4. Praxis

1 Intro

Themen heute

Karlsruher Institut für Technologie

Roter Faden: Objektorientierung

- (einfache) Vererbung
- virtual functions
- (einfache) Type-Casts in einer Klassen-Hierarchie
- mehr zu member access control (protected)
- Anwendung von Objektorientierung

Präsentation

Karlsruher Institut für Technologie

Praktisches Ziel der nächsten Workshops:
Ein Kommandozeilen-basiertes Zeichenprogramm

Präsentation

Praktisches Ziel der nächsten Workshops:
Ein Kommandozeilen-basiertes Zeichenprogramm

Theorie-Teil

- Live-Demo zur Vererbung (und mehr)
- Grundlegendes zum Bitmap-Framework

Karlsruher Institut für Technologie

Live-Demo (Sven)

Karlsruher Institut für Technologie

Live-Demo (Robert)

2 Vererbung
- Grundlegendes zur Vererbung
- Virtuelle Methoden
- Abstrakte Klassen
- dynamic_cast

Klassen und Instanzen

Karlsruher Institut für Technologie

Eine kurze Anmerkung zum Vokabular: Um den unsäglichen Begriff „Objekt“ aus diesem Kontext zunächst herauszuhalten, spreche ich von Instanzen. Analogie aus der Biologie: *Fuchur, Olis Hund,* ist eine Individuum der Unterart Haushund. Das Individuuen entspricht der Instanz, die Unterart einer Klasse.

Da es in C++ nur „Klassen“ (`class`, `struct`, `union`) und keine weitere Unterscheidung gibt wie Gattung, Art, Unterart, entsprechen die Klassen von C++ eigentlich allgemeineren Kategorien als nur etwa der Unterart.

Vererbung allgemein

Karlsruher Institut für Technologie

Eine Klasse kann von anderen Klassen erben. Wir beschränken uns vorläufig auf die einfachste Form von Vererbung, die sogenannte „public non-virtual single inheritance“. Zur Beschreibung wollen wir zwei Klassen verwenden: `CParent` und `CChild`.

Vererbung allgemein

Karlsruher Institut für Technologie

Eine Klasse kann von anderen Klassen erben. Wir beschränken uns vorläufig auf die einfachste Form von Vererbung, die sogenannte „public non-virtual single inheritance". Zur Beschreibung wollen wir zwei Klassen verwenden: `CParent` und `CChild`.

`CChild` erbe nun von `CParent` in der genannten einfachsten Weise. Dann wird eine Beziehung zwischen den Klassen hergestellt.

- Eine Analogie aus der Biologie: Jedes Säugetier ist ein Wirbeltier.
- Hier: Jede Instanz von `CChild` „ist" eine Instanz von `CParent` (eine „is-a"-Beziehung).

Vererbung allgemein

Karlsruher Institut für Technologie

Eine Klasse kann von anderen Klassen erben. Wir beschränken uns vorläufig auf die einfachste Form von Vererbung, die sogenannte „public non-virtual single inheritance". Zur Beschreibung wollen wir zwei Klassen verwenden: `CParent` und `CChild`.

`CChild` erbe nun von `CParent` in der genannten einfachsten Weise.

Dann wird eine Beziehung zwischen den Klassen hergestellt.

- Eine Analogie aus der Biologie: Jedes Säugetier ist ein Wirbeltier.
- Hier: Jede Instanz von `CChild` „ist" eine Instanz von `CParent` (eine „is-a"-Beziehung).

Diese Sprechweise ist etwas ungenau, sie meint: Eine Instanz von `CChild` hat *alle* Eigenschaften („data members") und Fähigkeiten („member functions") einer Instanz von `CParent`.

In der Biologie könnte man das ähnlich vollbringen: ich ordne Olis Hund aufgrund seiner Eigenschaften und Fähigkeiten der Unterart *Haushund* zu, damit ist Fuchur von der Art *Wolf*.

Beispiel mit konkreten Klassen

Karlsruher Institut für Technologie

```
class CParent
{
public:
        string name;
        int pos;

        void print();
        void set();

        virtual bool check();
        virtual void apply();
};
```

```
class CChild
        : public CParent
{
public:
        double rot;

        void turn();
        void set();

        virtual void apply();
};
```

Beispiel mit konkreten Klassen

Karlsruher Institut für Technologie

```
class CParent
{
public:
        string name;
        int pos;

        void print();
        void set();

        virtual bool check();
        virtual void apply();
};
```

```
class CChild
        : public CParent
{
public:
        double rot;

        void turn();
        void set();

        virtual void apply();
};
```

Wir verwenden im Folgenden zwei ≫Dinge≪:

```
CParent  myPrObj;
CChild   myObj;
```

Die Instanz myPrObj

Karlsruher Institut für Technologie

Wir wollen eine Instanz von `CParent` folgendermaßen visualisieren:

Zugriff auf member von myPrObj (1)

Karlsruher Institut für Technologie

Zugriffe auf die einfachen (nicht-virtuellen) member functions („Methoden“) und data members („Eigenschaften“) funktionieren wie gehabt:


```
myPrObj.name
```

Zugriff auf member von myPrObj (1)

Karlsruher Institut für Technologie

Zugriffe auf die einfachen (nicht-virtuellen) member functions („Methoden“) und data members („Eigenschaften“) funktionieren wie gehabt:

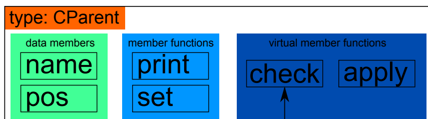
type: CParent

data members	member functions	virtual member functions
name	print	check
pos	set	apply

```
myPrObj.print()
```

Zugriff auf member von myPrObj (2)

Greifen wir über `myPrObj` auf eine virtual member function zu, so bleibt alles wie gehabt:

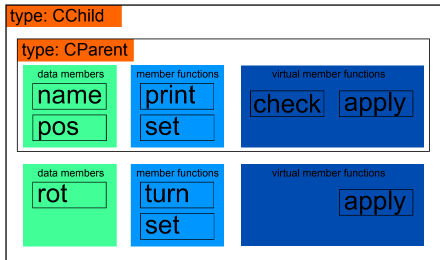


Die Instanz myObj

Vergessen wir ab nun die Instanz (das ≫Ding≪) `myPrObj`, sie sei hiermit entlassen.

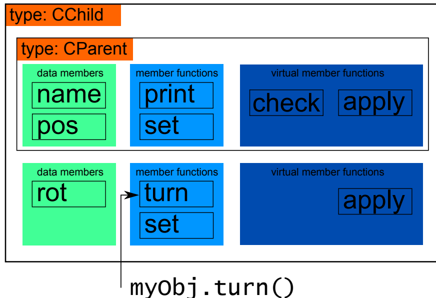
Wir wollen eine Instanz von `CChild` wie folgt visualisieren. Zu sehen ist auch das sog. „base class subobject“* von der Basisklasse CParent.

*Das ist leider etwas verwirrend, denn ein „object“ nach dem Standard haben wir bislang als ≫Ding≪ bezeichnet. Ein ≫Ding≪ enthält jedoch keine Funktionen. Ich hoffe es ist dennoch klar, dass mit dem Bild die „is-a“-Beziehung zwischen Instanzen von CChild und Instanzen von CParent zum Ausdruck gebracht werden soll.



Zugriff auf member von myObj (1)

Zugriffe auf die einfachen (nicht-virtuellen) member functions und data members funktionieren auch hier wie gehabt:



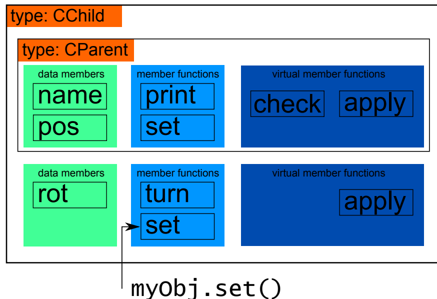
Zugriff auf member von myPrObj (2) - name hiding

Karlsruher Institut für Technologie

name hiding (Standard 3.3.7)

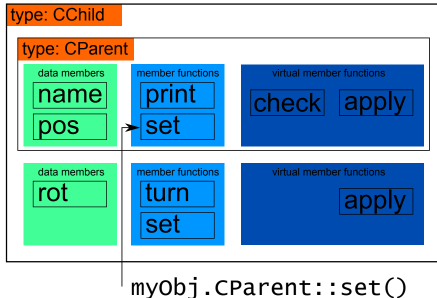
Hat eine member function oder ein data member einer Kindklasse denselben Namen wie eine member function oder ein data member einer Basisklasse (hier: `set`), so wird der Name aus der Basisklasse versteckt.

Etwas anschaulicher lässt es sich als „Überdecken“ beschreiben: Das Element aus der Kindklasse überdeckt den Zugriff auf das Element der Basisklasse:



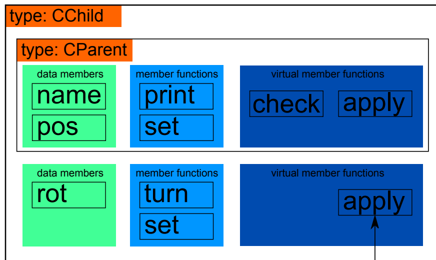
Zugriff auf member von myPrObj (3) - qualified-id

Detail (wird wirklich wirklich rar verwendet):
Man kann dennoch auf das Element aus der Basisklasse zugreifen, und zwar mittels einer sog. qualified-id:



Zugriff auf member von myObj (4)

Zugriffe auf virtual member functions funktionieren über `myObj` effektiv genauso wie Zugriffe auf normale Funktionen, allerdings ist der Mechanismus nicht das name hiding (später mehr).



Pointer und Instanzen

KIT – Karlsruher Institut für Technologie

Wir können freilich einen Pointer auf `myObj` anlegen:

```
CChild* pMyObj = &myObj;
```

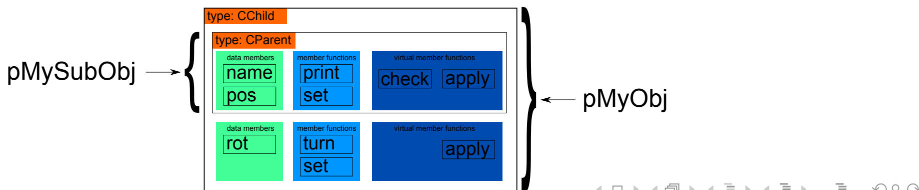
Pointer und Instanzen

Wir können freilich einen Pointer auf `myObj` anlegen:

```
CChild* pMyObj = &myObj;
```

`myObj` enthält jedoch *zudem* ein subobject von der Klasse `CParent`. Dieses subobject hat als »Ding« ebenfalls eine Adresse, man erhält sie durch einen „implicit type cast“ (Standard, 4.10:3):

`CParent* pMySubObj = &myObj;` (oder auch `pMySubObj = pMyObj;`)



static type

static type (Standard, 1.3.11)

Der Typ (eines ≫Dings≪ oder einer expression), so wie der Compiler ihn sieht. D.h. ohne Effekte zu berücksichtigen, die zur Laufzeit auftreten.

Beispiel:

`int i;`	`i` hat den static type `int`
`int* pi;`	`pi` hat den static type `int*`
`*pi`	hat den static type `int`
`*pMySubObj`	hat den static type `CParent`

dynamic type

Karlsruher Institut für Technologie

dynamic type (Standard, 1.3.3)

blabla lvalue blabla (unmittelbar unverständlich)

Beispiel:

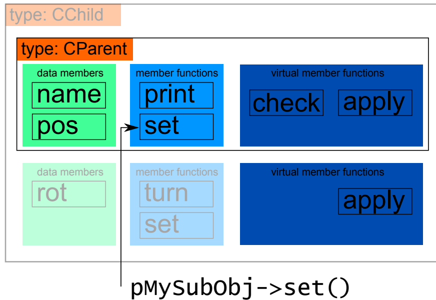
`int i;`	`i` hat den dynamic type `int`
`int* pi;`	`pi` hat den dynamic type `int*`
`*pi`	hat den dynamic type `int`
`*pMySubObj`	hat den dynamic type `CChild`

Der dynamic type ist der Typ des meist-abgeleiteten (ableiten $\Longrightarrow$ Vererbung) »Dings«, auf dessen subobject ein Pointer verweist. In unserer Visualisierung entspricht dies dem Typ der übergeordnetsten Instanz.

Zugriff mittels pMySubObj (1)

Karlsruher Institut für Technologie

Nutzt man `pMySubObj`, um auf data members oder *non-virtual* member functions zuzugreifen, wird der static type verwendet. Somit ist das übergeordnete des subobjects sozusagen unsichtbar. Dementsprechend wird beim Zugriff über `pMySubObj` die zweite Deklaration von `set` „nicht gesehen“.



Overriding

Karlsruher Institut für Technologie

Eine Unzulänglichkeit der deutschen Sprache: Es gibt Überladen (overload) und Überschreiben (overwrite), aber mir fällt keine gute (eindeutige) Übersetzung für *override* ein. In der deutschen Informatik verwendet man für override tatsächlich überschreiben, aber das klingt in meinen Ohren zu sehr nach Verdecken (name hiding).

Overriding (Standard, 10.3:2)

In einer Klasse `CParent` sei eine virtual member function `apply` deklariert. Eine Klasse `CChild` erbe direkt oder indirekt von `CParent`. Wenn nun in `CChild` eine Funktion mit *demselben Namen und derselben Parameter-Liste* wie `CParent::apply` deklariert ist, so ist `CChild::apply` ebenfalls virtual (egal, ob sie so explizit deklariert wurde oder nicht) und sie *overrides* `CParent::apply`.

Das Aufrufen virtueller Funktionen (1)

Karlsruher Institut für Technologie

calling a virtual function (Standard, 5.2.2:1)

Wird eine virtual function aufgerufen, wird der dynamic type des ≫Dings≪ herangezogen. Es wird dann ausgehend vom dynamic type in der Vererbungshierarchie in Richtung der Elter-Klassen nach einem override der Funktion gesucht (und der erste Treffer verwendet).

Das Aufrufen virtueller Funktionen (1)

Karlsruher Institut für Technologie

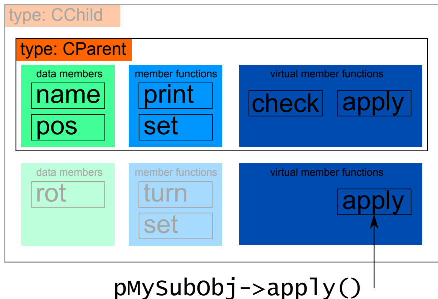
calling a virtual function (Standard, 5.2.2:1)

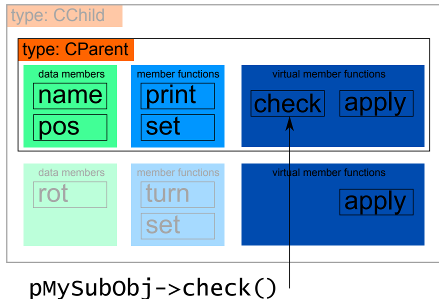
Wird eine virtual function aufgerufen, wird der dynamic type des »Dings« herangezogen. Es wird dann ausgehend vom dynamic type in der Vererbungshierarchie in Richtung der Elter-Klassen nach einem override der Funktion gesucht (und der erste Treffer verwendet).

Normalerweise (bei nicht-virtuellen Funktionen) wird vom static type ausgegangen, daher ruft `pMySubObj->set();` auch `CParent::set` auf. Bei virtual function calls hingegen wird vom *dynamic type* ausgegangen.

Da der dynamic type von `*pMySubObj` eben `CChild` ist, ruft `pMySubObj->apply();` dann `CChild::apply` auf.

Das Aufrufen virtueller Funktionen (2)





Der virtuelle Destruktor

Karlsruher Institut für Technologie

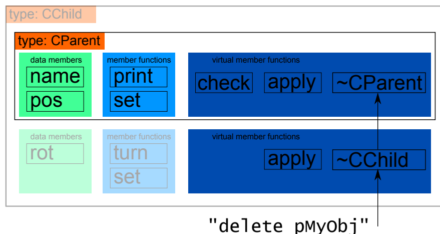
Angenommen, ich habe nun `pMyObj = new CChild;` geschrieben. Was macht dann `delete pMyObj;`?

Der virtuelle Destruktor

Angenommen, ich habe nun `pMyObj = new CChild;` geschrieben. Was macht dann `delete pMyObj;`?

Antwort: Das kommt darauf an, ob der Destruktor von `CParent` virtuell ist!

- Ist der Destruktor *nicht* virtuell, so wird nur `pMyObj->~CParent()` aufgerufen.
- Ist der Destruktor virtuell, so wird zunächst `pMyObj->~CChild()` und anschließend `pMyObj->~CParent()` aufgerufen.



Der virtuelle Destruktor, allgemein

Der virtuelle Destruktor-Aufruf funktioniert ähnlich wie der Aufruf einer normalen virtuellen Funktion, mit zwei Ausnahmen:

- Der Destruktor (dtor) hat in jeder Klasse einen eigenen Namen (`~Classname`).
- Es werden *alle* overridden Destruktoren (die der Basisklassen) aufgerufen, und zwar beginnend mit dem most-derived (final overrider), dann jeweils bei Ende des dtors einer Klasse der dtor der direkten Elterklasse dieser Klasse.

Der virtuelle Destruktor, allgemein

Karlsruher Institut für Technologie

Der virtuelle Destruktor-Aufruf funktioniert ähnlich wie der Aufruf einer normalen virtuellen Funktion, mit zwei Ausnahmen:

- Der Destruktor (dtor) hat in jeder Klasse einen eigenen Namen (`~Classname`).
- Es werden *alle* overridden Destruktoren (die der Basisklassen) aufgerufen, und zwar beginnend mit dem most-derived (final overrider), dann jeweils bei Ende des dtors einer Klasse der dtor der direkten Elterklasse dieser Klasse.

Zusätzlich zum dtor-Aufruf wird natürlich noch der Speicher freigegeben. Dabei spielt es keine Rolle, ob der dtor virtuell ist oder nicht.

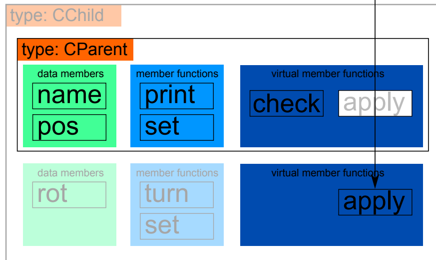
pure virtual member functions

Karlsruher Institut für Technologie

Bei einer virtual member function kann die Implementierung komplett weggelassen werden, indem hinter die Funktions-Deklaration ein `= 0;` geschrieben wird. Sie ist dann nur noch eine leere Hülle. Nanu?

pure virtual member functions

Bei einer virtual member function kann die Implementierung komplett weggelassen werden, indem hinter die Funktions-Deklaration ein `= 0;` geschrieben wird. Sie ist dann nur noch eine leere Hülle. Nanu?
Die Idee dabei ist: Pointer auf die Basisklasse können die pure virtual member function aufrufen, die Implementierung (der Inhalt) dieser Funktion wird dann durch abgeleitete Klassen bereitgestellt.



Abstrakte Klassen

Karlsruher Institut für Technologie

Es gibt einen Hauptunterschied zwischen: `virtual void apply() { }` und `virtual void apply() = 0;` Dieser ist, dass eine einzige pure virtual member function die Klasse zu einer sog. *abstrakten Klasse* macht.

Abstrakte Klassen

KIT Karlsruher Institut für Technologie

Es gibt einen Hauptunterschied zwischen: `virtual void apply() { }` und `virtual void apply() = 0;` Dieser ist, dass eine einzige pure virtual member function die Klasse zu einer sog. *abstrakten Klasse* macht.

Abstrakte Klassen (Standard, 10.4:1-2)

Eine *abstrakte Klasse* ist eine Klasse die nur als Basisklasse für andere Klassen verwendet werden kann; es können keine Instanzen einer abstrakten Klasse erzeugt werden (nur base class subobjects). Eine Klasse ist eine abstrakte Klasse wenn sie mindestens eine pure virtual member function hat. Beachte: Sie kann diese geerbt haben (wenn noch kein override in der Hierarchie existiert).

Interface

Karlsruher Institut für Technologie

Ein abstrakte Klasse, die *ausschließlich* deren member ausschließlich pure virtual member functions sind, kann nennt man auch Interface.

Interface

Karlsruher Institut für Technologie

Ein abstrakte Klasse, die *ausschließlich* deren member ausschließlich pure virtual member functions sind, kann nennt man auch Interface.

Interfaces werden für sehr viele Dinge verwendet und sind neben dem Arbeiten mit Objekten ein Hauptaspekt im objektorientierten Programmieren. Man definiert mittels Interfaces bspw. die gemeinsame „Sprache“, mit der zwei voneinander unabhängige Komponenten interagieren können. Die eine Seite nutzt dabei einen Pointer vom Typ `MyInterface*` und die andere Seite *implementiert* das Interface, d.h. hat von eine Klasse, die von `MyInterface` erbt und die pure virtual member functions implementiert. Die letztere Seite erzeugt dann eine Instanz dieser abgeleiteten Klasse und gibt der ersten Seite einen Pointer.

Der Vollständigkeit halber: dynamic_cast

Karlsruher Institut für Technologie

Man kann implizit von einem `CChild*` einen `CParent*` erhalten. Umgekehrt geht das nicht ohne weiteres, denn z.B. ein Pointer auf `myPrObj` (es ist untot! xD) kann nicht sinnvollerweise in einen Pointer auf `CChild*` verwandelt werden (z.B.: es fehlt die Information des data members `rot`).

Der Vollständigkeit halber: dynamic_cast

Karlsruher Institut für Technologie

Man kann implizit von einem `CChild*` einen `CParent*` erhalten. Umgekehrt geht das nicht ohne weiteres, denn z.B. ein Pointer auf `myPrObj` (es ist untot! xD) kann nicht sinnvollerweise in einen Pointer auf `CChild*` verwandelt werden (z.B.: es fehlt die Information des data members `rot`).

Wenn man schreibt `dynamic_cast < CChild* > ( pMySubObj )`, so ist das Resultat dieser expression abhängig vom *dynamic type* von `*pMySubObj` – verweist `pMySubObj` auf ein base class subobject einer Instanz von `CChild`, so ist das Ergebnis ein gültiger Pointer auf diese Instanz (auf das ≫Ding≪). Anderenfalls ist das Resultat `0` (ein ungültiger Pointer).

3. Das Bitmap-Framework
   - Bitmap24 und Algorithmen
   - BatchBitmap24 und Zeichen-Objekte

Die Klasse Bitmap24

Karlsruher Institut für Technologie

Die Klasse `Bitmap24` ist eine minimale Klasse zum Abspeichern von Bitmaps (.bmp). Sie enthält dafür die member function `save`, der am einfachsten ein String-Literal als Parameter übergeben wird, etwa `save("filename.bmp")`.

Die Klasse Bitmap24

KIT
Karlsruher Institut für Technologie

Die Klasse `Bitmap24` ist eine minimale Klasse zum Abspeichern von Bitmaps (.bmp). Sie enthält dafür die member function `save`, der am einfachsten ein String-Literal als Parameter übergeben wird, etwa `save("filename.bmp")`.

Wie speichere ich eine Bitmap?

Die Bilddaten (Pixel) müssen in den Puffer von `Bitmap24` liegen, sie können dorthin mit `setPixel` geschrieben werden. Anschließend speichert ein Aufruf der member function `save` die Bitmap als Datei. Geläufige C++ Implementierungen werden diese Datei in das Arbeitsverzeichnis legen, etwa im Ordner der ausführbaren Datei. Ebenfalls bei den geläufigen Implementierungen kann man dem Dateinamen noch einen Pfad voranstellen.

Objektorientierung im Bitmap-Framework

Das Bitmap-Framework legt den Wert auf die Interaktion von Instanzen (Objekten). Zum Zeichnen einer Line soll daher ein Linien-Objekt erstellt werden, welches dann mit einem „Leinwand“-Objekt interagiert.

Objektorientierung im Bitmap-Framework

Das Bitmap-Framework legt den Wert auf die Interaktion von Instanzen (Objekten). Zum Zeichnen einer Line soll daher ein Linien-Objekt erstellt werden, welches dann mit einem „Leinwand“-Objekt interagiert.

Damit die beiden Objekte miteinander interagieren können, brauchen sie sozusagen eine gemeinsame Sprache. Hier basiert diese auf der Verwendung von Pixeln: Die Linie weiß, wie sie in Pixeln aussieht, und die Leinwand stellt eine Möglichkeit zur Verfügung, pixelweise auf sie zu zeichnen.

Minimalimplementierung der Leinwand

Karlsruher Institut für Technologie

Wir werden später sehen, dass es eigentlich vorteilhaft ist, die gemeinsame Sprache zwischen Zeichenobjekten (Linie, Kreis usw.) und Leinwand abstrakt in Form eines Interface festzuhalten. Wir wollen jedoch zunächst unser Programm einfach halten und Abstraktion vermeiden.

Minimalimplementierung der Leinwand

Karlsruher Institut für Technologie

Wir werden später sehen, dass es eigentlich vorteilhaft ist, die gemeinsame Sprache zwischen Zeichenobjekten (Linie, Kreis usw.) und Leinwand abstrakt in Form eines Interface festzuhalten. Wir wollen jedoch zunächst unser Programm einfach halten und Abstraktion vermeiden.

Die `Bitmap24`-Klasse, die eigentlich zum Abspeichern einer Bitmap-Datei dient, reicht aus, um als Leinwand-Klasse verwendet werden zu können: Man kann pixelweise auf das hinterlegte Bitmap zugreifen, dies entspricht einem pixelweisen Zeichnen (`setPixel`).

Zeichen-Algorithmus

Karlsruher Institut für Technologie

Bevor man damit beginnt, eine Klasse für ein Zeichen-Objekt zu schreiben, sollte man eine Idee haben, wie das zu zeichnende Objekt in Pixeln aussieht. Man benötigt also eine Vorgehensweise, eine Menge von Pixeln zu „setzen" (den Farbwert zu setzen) $\Longrightarrow$ ein Algorithmus.

Zeichen-Algorithmus

Karlsruher Institut für Technologie

Bevor man damit beginnt, eine Klasse für ein Zeichen-Objekt zu schreiben, sollte man eine Idee haben, wie das zu zeichnende Objekt in Pixeln aussieht. Man benötigt also eine Vorgehensweise, eine Menge von Pixeln zu „setzen“ (den Farbwert zu setzen) $\Longrightarrow$ ein Algorithmus.

Der einfachste Fall hierfür ist der Algorithmus zum Füllen der gesamten Leinwand mit einer Farbe. Es muss hierbei lediglich jedem Pixel diese Farbe zugewiesen werden, der Algorithmus benötigt also als Eingabe eine Farbe sowie die Leinwand, auf der er agieren soll.

Implementierung eines Zeichen-Algorithmus (1)

Karlsruher Institut für Technologie

In unserem einfachen Programm können wir nun eine Funktion schreiben:

```
void fillCanvas(Bitmap24& targetCanvas, Color fillColor);
```

Beachte: Wir müssen die Leinwand als Referenz übergeben, da der Algorithmus ja auf dem übergebenen »Ding« arbeiten soll, und nicht auf einer Kopie (= nicht auf einem neuen »Ding«).

Implementierung eines Zeichen-Algorithmus (2)

Karlsruher Institut für Technologie

Es stehen nun innerhalb der Funktion `fillCanvas` die Eigenschaften und Fähigkeiten des `Bitmap24` zur Verfügung, wir können also pixelweise die Bitmap schreiben und die Höhe sowie Breite abfragen. Da die Pixel bei $(0, 0)$ beginnen und die Leinwand bei $(\text{width} - 1, \text{height} - 1)$ zu Ende ist, sind alle nötigen Informationen für das Füllen der Leinwand vorhanden.

Zeichen-Objekte

Karlsruher Institut für Technologie

Wir könnten basierend auf dem Algorithmus, der oben beispielhaft als Funktion designed wurde, bereits ein Objekt (eine Klasse) machen. So könnte dieses Objekt bspw. die Farbe als data member speichern und eine member function enthalten, die als Parameter nur noch die Leinwand annehmen muss. Was wäre der Vorteil davon?

Zeichen-Objekte

Karlsruher Institut für Technologie

Wir könnten basierend auf dem Algorithmus, der oben beispielhaft als Funktion designed wurde, bereits ein Objekt (eine Klasse) machen. So könnte dieses Objekt bspw. die Farbe als data member speichern und eine member function enthalten, die als Parameter nur noch die Leinwand annehmen muss. Was wäre der Vorteil davon?

Z.B. könnte man ein solches Objekt im Voraus erzeugen, und anschließend auf eine Reihe von Leinwänden anwenden. So könnte man etwa ein Zeichenobjekt pro Grundfarbe anlegen. Aber richtig groß ist der Vorteil noch nicht, das ändert sich aber mit ein paar mehr Schritten.

Klassifizierung

Karlsruher Institut für Technologie

Wenn ich die Menge von Klassen für Zeichenobjekte betrachte, so fallen mir zwei Punkte auf:

- Die Zeichenobjekte entsprechen ungefähr Werkzeugen in der Realität (betrachte auch den Werkzeugkasten im Zeichenprogramm deiner Wahl, etwa m$PAIN).
- Die Zeichenobjekte sind alle Zeichenobjekte (triviale Aussage? nein!)

Klassifizierung

Karlsruher Institut für Technologie

Wenn ich die Menge von Klassen für Zeichenobjekte betrachte, so fallen mir zwei Punkte auf:

- Die Zeichenobjekte entsprechen ungefähr Werkzeugen in der Realität (betrachte auch den Werkzeugkasten im Zeichenprogramm deiner Wahl, etwa m$PAIN).
- Die Zeichenobjekte sind alle Zeichenobjekte (triviale Aussage? nein!)

Die Klassen der Zeichenobjekte sind unabhängig voneinander, sie sind aber alle insofern ähnlich, dass sie mit einer Leinwand interagieren (= sie sind alle Zeichenobjekte). Man kann also die Menge von Klassen (für Linien, Kreise, Rechtecke usw.) kategorisieren bzw. klassifizieren. Dafür verwenden wir eine gemeinsame Basis-Klasse (alle Zeichenobjekt-Klassen erben von derselben Basisklasse).

Interfaces

Karlsruher Institut für Technologie

Da die Zeichenobjekt-Klassen voneinander unabhängig sind (z.B. um einen Kreis zu zeichnen hilft es normalerweise nicht weiter, zu wissen, wie man eine gerade Linie zeichnet), ist ihre einzige Gemeinsamkeit die Anwendung auf eine Leinwand. Dies kann man zum Ausdruck bringen, indem man einer gemeinsamen Basisklasse nur eine einzige pure virtual member function verpasst (und sonst keine member).

Interfaces

Karlsruher Institut für Technologie

Da die Zeichenobjekt-Klassen voneinander unabhängig sind (z.B. um einen Kreis zu zeichnen hilft es normalerweise nicht weiter, zu wissen, wie man eine gerade Linie zeichnet), ist ihre einzige Gemeinsamkeit die Anwendung auf eine Leinwand. Dies kann man zum Ausdruck bringen, indem man einer gemeinsamen Basisklasse nur eine einzige pure virtual member function verpasst (und sonst keine member).

Diese Funktion dient dann dazu, ein Zeichenobjekt auf eine Leinwand anzuwenden, könnte also `apply` genannt werden und ein `Bitmap24&` als Parameter annehmen.

Ein Ausblick: Kommandozeile

Karlsruher Institut für Technologie

In einem nächsten Schritt (Workshop) soll von der Kommandozeile aus gezeichnet werden können, also etwa durch eine Eingabe:
`moveto 0.5 0.4<ENTER>lineto 0 0<ENTER>`.

Der Benutzer muss hierbei um eine Linie zu zeichnen nur einen Endpunkt angeben, die Linie wird dann von einer vorhandenen, aktuellen Position bis zur angegebenen Position gezeichnet und anschließend die Endposition als neue „aktuelle Position“ gespeichert. Auf diese Weise sind auch Polygone sehr leicht zu zeichnen. Diese aktuelle Position sowie eine aktuelle Farbe – zusammen sozusagen ein Pinsel – müssen also noch irgendwo gespeichert werden. Eine Möglichkeit hierfür ist, die Leinwand-Klasse zu erweitern. Man kann sich darunter dann etwa einen Tintenstrahldrucker oder Plotter vorstellen.

Die Klasse BatchBitmap24

Karlsruher Institut für Technologie

Die Zeichenobjekte, die also theoretisch auf einer Instanz eines `Bitmap24` arbeiten können, sollten – um die Benutzereingabe auf der Kommandozeile zu vereinfachen – diesen vorhandenen Pinsel verwenden.

Die Klasse BatchBitmap24

Karlsruher Institut für Technologie

Die Zeichenobjekte, die also theoretisch auf einer Instanz eines `Bitmap24` arbeiten können, sollten – um die Benutzereingabe auf der Kommandozeile zu vereinfachen – diesen vorhandenen Pinsel verwenden.

Die Klasse `BatchBitmap24` ist eine kleine Erweiterung von `Bitmap24`, die genau diesen Pinsel speichert. Daher benötigen die Zeichenobjekte als Interaktionsobjekt eine Instanz eines `BatchBitmap24`.

Die unsäglichen Koordinaten-Klassen

Karlsruher Institut für Technologie

Mit etwas zu viel Beschützerinstinkt habe ich wohl die beiden Klassen `AbsoluteCoordinate` und `RelativCoordinate` geschaffen.

Die unsäglichen Koordinaten-Klassen

Mit etwas zu viel Beschützerinstinkt habe ich wohl die beiden Klassen `AbsoluteCoordinate` und `RelativCoordinate` geschaffen.

AbsoluteCoordinate

Fasst egtl. nur den x- und y-Wert einer Koordinate zusammen in eine Datenstruktur. Der Beschützerinstinkt drückt sich dabei so aus, dass eine Instanz von `AbsoluteCoordinate` immer einer Leinwand zugeordnet ist. Das verhindert (soll verhindern), dass die Koordinate auf eine ungültige Position verweist (Lokalität von Fehlermeldungen).

Die unsäglichen Koordinaten-Klassen

Karlsruher Institut für Technologie

Mit etwas zu viel Beschützerinstinkt habe ich wohl die beiden Klassen `AbsoluteCoordinate` und `RelativCoordinate` geschaffen.

AbsoluteCoordinate

Fasst egtl. nur den x- und y-Wert einer Koordinate zusammen in eine Datenstruktur. Der Beschützerinstinkt drückt sich dabei so aus, dass eine Instanz von `AbsoluteCoordinate` immer einer Leinwand zugeordnet ist. Das verhindert (soll verhindern), dass die Koordinate auf eine ungültige Position verweist (Lokalität von Fehlermeldungen).

RelativeCoordinate

Die `RelativeCoordinate` ist dann sozusagen die „freie Variante einer sicheren Koordinate“. Sie speichert eine Koordinate als Vielfaches von Breite bzw. Höhe, ihre Komponenten liegen also in $[0, 1]$ (dass die 1 eingeschlossen werden sollte, werde ich vielleicht einmal an anderer Stelle erläutern). Dadurch, dass sie unabhängig ist von einer konkreten Leinwand (sowohl durch die fehlende Prüfung aber logisch auch da sie sich eben *nicht* auf eine feste Breite oder Höhe bezieht) eignet sie sich in meinen Augen wesentlich besser etwa um in einem Linien-Zeichenobjekt eine Endkoordinate zu speichern.

Das Interface IBatchDrawable

Karlsruher Institut für Technologie

Dieses Interface definiert nun die besprochene pure virtual member function, welche alle Zeichenobjekte (deren Klassen) auszeichnet:

```
virtual bool applyTo( BatchBitmap24& ) = 0;
```

Der Rückgabewert soll hierbei `true` im Falle des erfolgreichen Zeichnens sein.

Das Interface IBatchDrawable

Karlsruher Institut für Technologie

Dieses Interface definiert nun die besprochene pure virtual member function, welche alle Zeichenobjekte (deren Klassen) auszeichnet:

```
virtual bool applyTo( BatchBitmap24& ) = 0;
```

Der Rückgabewert soll hierbei `true` im Falle des erfolgreichen Zeichnens sein.

Die Klassen der Zeichenobjekte erben also von `IBatchDrawable`, und override die pure virtual member function. Im override rufen sie den Algorithmus auf und zeichnen somit auf die Leinwand.

Ausblick: Der Sinn von IBatchDrawable

Karlsruher Institut für Technologie

- Man kann über einen `IBatchDrawable*`-Pointer die Zeichenobjekte auf eine Instanz eines `BatchBitmap24` anwenden, ohne näheres über konkrete Zeichenobjekt (den static type) zu wissen. Mittels dynamic storage duration / `new` lassen sich somit zur Laufzeit verschiedene Zeichenobjekte erstellen (bspw. auf Anfrage des Benutzers!) und allgemein auf eine Leinwand anwenden.
- Die Klassifizierung aller Zeichenobjekte als „is-a“ `IBatchDrawable` erlaubt es, eine Menge von Zeichenobjekten gemeinsam zu „speichern“ – gemeint ist etwa ein Array von `IBatchDrawable*`-Pointern, oder der bekannte Ringpuffer mit `IBatchDrawable*` statt `double`.

4 Praxis

Praxis!

Karlsruher Institut für Technologie

`https://github.com/kit-cpp-workshop/workshop-ss12-03`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`