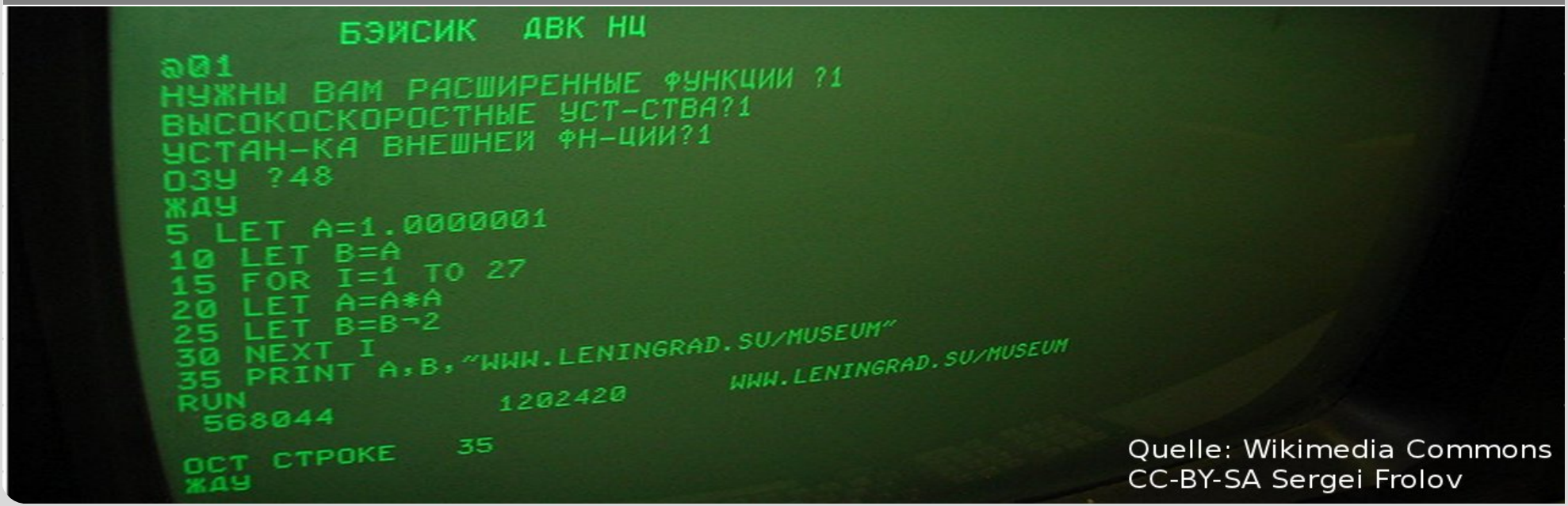
Karlsruher Institut für Technologie

C++ Workshop

Addendum zum 3. Block: Vererbung, 18.05.2012

Robert Schneider, Markus Jung | 23. Mai 2012



Gliederung

1 Grundlegendes zur Vererbung

Klassen und Instanzen

Eine kurze Anmerkung zum Vokabular: Um den unsäglichen Begriff „Objekt“ aus diesem Kontext zunächst herauszuhalten, spreche ich von Instanzen.
Eine Instanz ist ein konkretes Individuum, das nach dem Plan der Klasse erstellt wurde. Sie hat damit die Eigenschaften und Fähigkeiten, welche in der Klasse beschrieben werden.
Analogie aus der Biologie: *Fuchur, Olis Hund,* ist ein konkretes Individuum der Unterart Haushund. Er kann laufen (Fähigkeit) und zwar mit einer nur ihm eigenen Geschwindigkeit (Eigenschaft).

Natürlich hat dieser Vergleich mit der Biologie seine Grenzen. Eine davon ist etwa, dass es dort nicht nur Unterarten gibt, sondern mehrere Ebenen von Kategorien, etwa Gattungen und Arten. In C++ hingegen gibt es nur eine Art von Kategorie: die Klasse (`class`, `struct`, `union`).

Vererbung allgemein

Eine Klasse kann von anderen Klassen abgeleitet werden. 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

Eine Klasse kann von anderen Klassen abgeleitet werden. 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` sei nun von `CParent` in der genannten einfachsten Weise abgeleitet. Dann wird eine Beziehung zwischen den Klassen hergestellt.

- Eine Analogie aus der Biologie: Die Kategorie „Säugetier“ ist eine Unterkategorie von „Wirbeltier“.
- Hier: `CChild` ist eine Erweiterung von `CParent`.

Vererbung allgemein

Eine Klasse kann von anderen Klassen abgeleitet werden. 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` sei nun von `CParent` in der genannten einfachsten Weise abgeleitet. Dann wird eine Beziehung zwischen den Klassen hergestellt.

- Eine Analogie aus der Biologie: Die Kategorie „Säugetier“ ist eine Unterkategorie von „Wirbeltier“.
- Hier: `CChild` ist eine Erweiterung von `CParent`.

Diese Sprechweise ist etwas ungenau, sie meint: Eine Instanz von `CChild` hat *alle* Eigenschaften („data members“) und Fähigkeiten („member functions“) einer Instanz von `CParent`, sie *ist auch eine Instanz der Klasse* `CParent` (eine „is-a“-Beziehung).
In der Biologie könnte man das ähnlich vollbringen: Olis Hund ist von der Unterart *Haushund* und von der Art *Wolf*.

Beispiel mit konkreten Klassen

```
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

```
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;
```

Nochmal Sprechweise

In unserem Beispiel ist `CChild` eine „Kindklasse“ von `CParent`, oder andersherum: `CParent` ist eine „Elterklasse“. Man spricht auch von „Basisklasse“ (Elter) und „abgeleiteter Klasse“ (Kind).

Nochmal Sprechweise

In unserem Beispiel ist `CChild` eine „Kindklasse“ von `CParent`, oder andersherum: `CParent` ist eine „Elterklasse“. Man spricht auch von „Basisklasse“ (Elter) und „abgeleiteter Klasse“ (Kind).

Statt von „Instanzen“ spricht man häufig auch von „Objekten“. Der Begriff „Instanz“ geht dabei mehr auf den Bezug zur Klasse ein (eine Instanz einer Klasse), während der Begriff „Objekt“ tendenziell eher auf die Zusammenfassung von Eigenschaften und Fähigkeiten und die eigenständige Existenz eingeht (es existiert selbst als Objekt, nicht nur als Bündel).

Nochmal Sprechweise

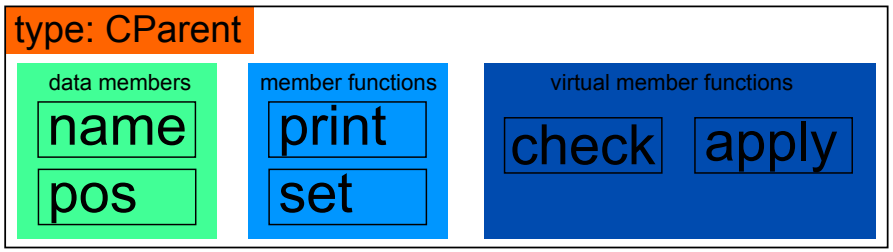
In unserem Beispiel ist `CChild` eine „Kindklasse“ von `CParent`, oder andersherum: `CParent` ist eine „Elterklasse“. Man spricht auch von „Basisklasse“ (Elter) und „abgeleiteter Klasse“ (Kind).

Statt von „Instanzen“ spricht man häufig auch von „Objekten“. Der Begriff „Instanz“ geht dabei mehr auf den Bezug zur Klasse ein (eine Instanz einer Klasse), während der Begriff „Objekt“ tendenziell eher auf die Zusammenfassung von Eigenschaften und Fähigkeiten und die eigenständige Existenz eingeht (es existiert selbst als Objekt, nicht nur als Bündel).

Der Begriff des ≫Dinges≪ ist hiervon verschieden, da das ≫Ding≪ ein Speicherbereich ist. Es hat somit eigentlich keine member functions, diese Stecken in seinem Typen. Zudem bezeichnet in der expression `int foo;` der Name `foo` ein ≫Ding≪, man würde dies in C++ aber weder eine Instanz noch ein Objekt nennen, da es keine Klasse `int` gibt.

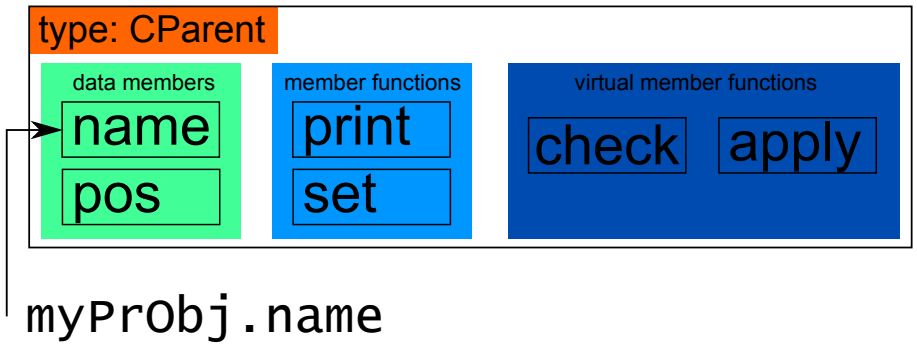
Die Instanz myPrObj

Wir wollen eine Instanz von `CParent` folgendermaßen visualisieren:



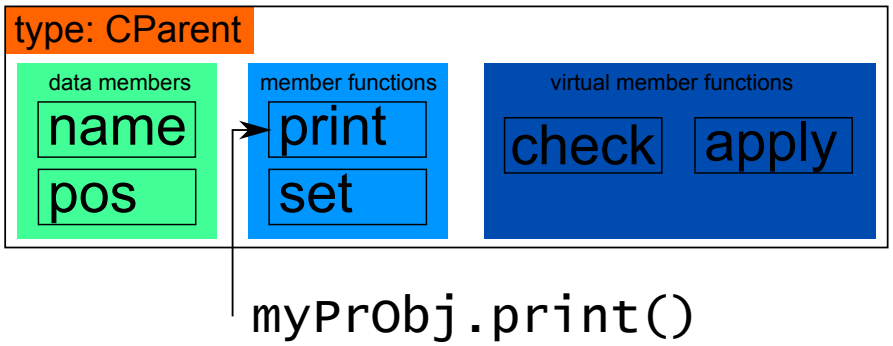
Zugriff auf member von myPrObj (1)

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



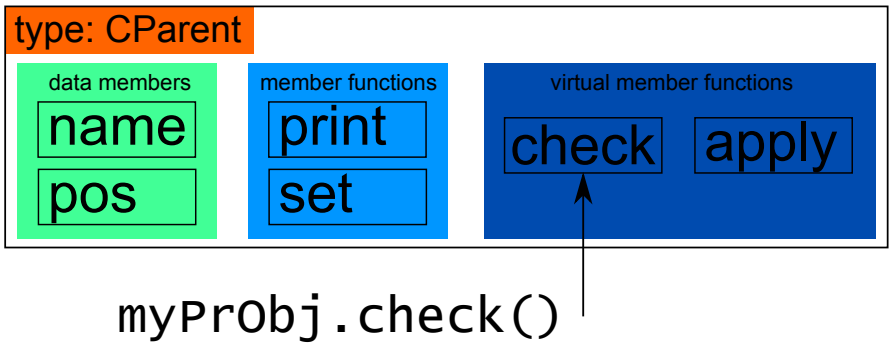
Zugriff auf member von myPrObj (1)

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



Zugriff auf member von myPrObj (2)

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

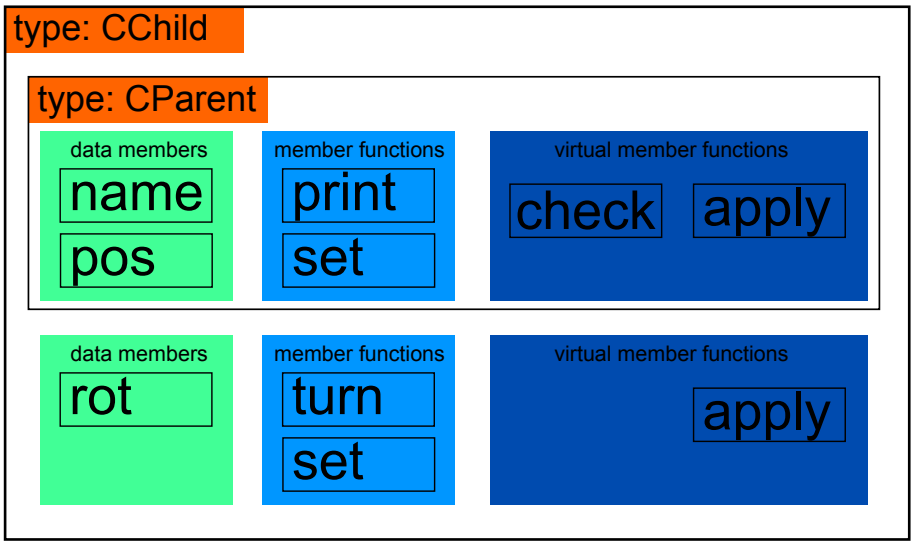


Die Instanz myObj

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

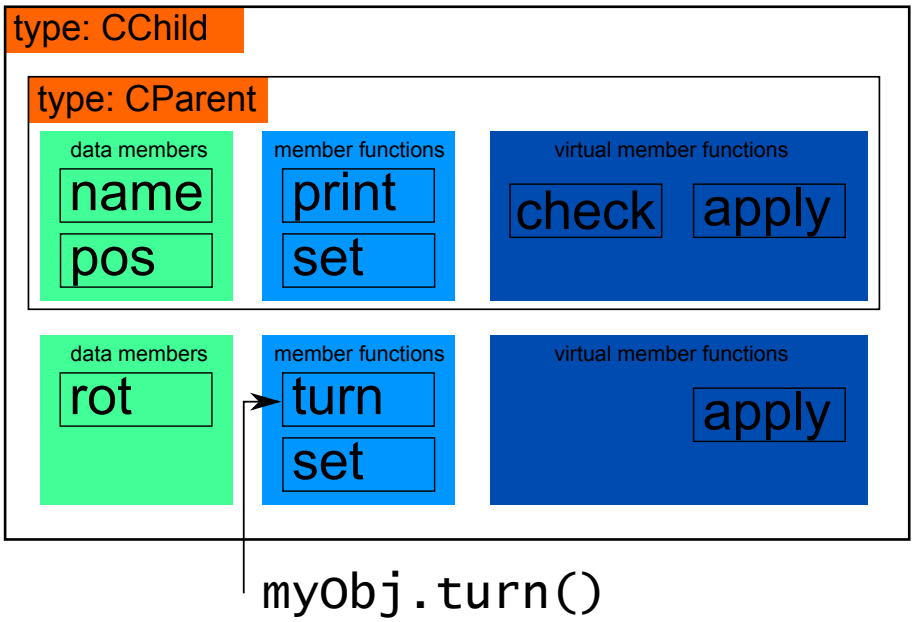
Wir wollen eine Instanz von `CChild` wie folgt visualisieren. Zu sehen ist auch das sog. „base class subobject“* von der Basisklasse CParent.

*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:

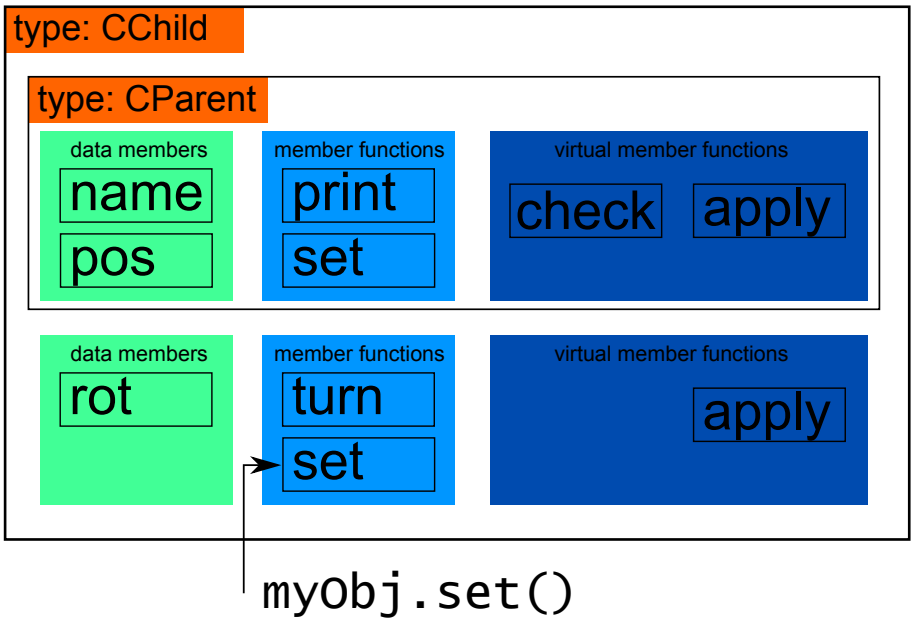


Zugriff auf member von myPrObj (2) - name hiding

name hiding (Standard 3.3.7)

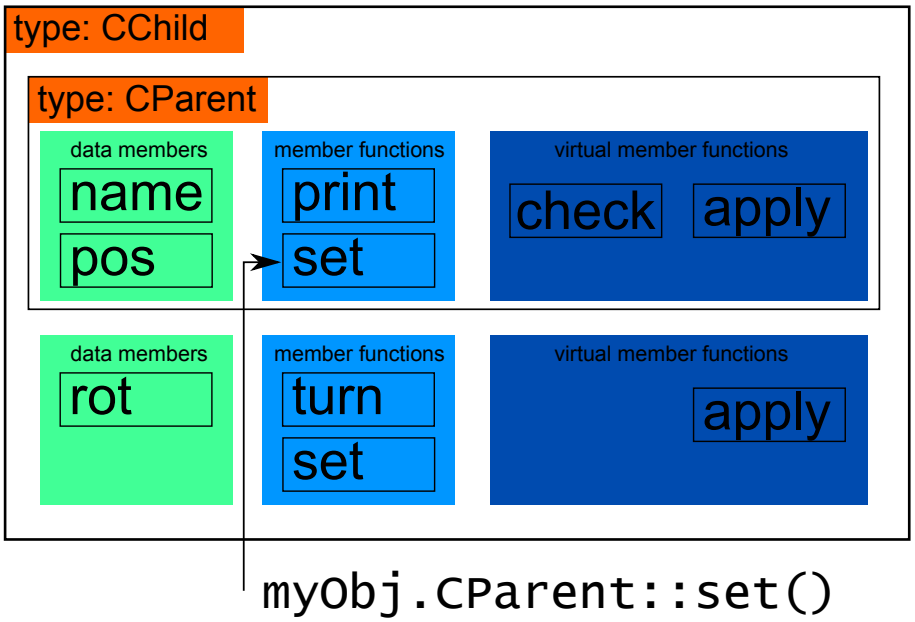
Hat eine member function oder ein data member einer Kindklasse denselben Namen wie eine member function oder ein data member einer Basisklasse (hier: `set`), so wird der Name aus der Basisklasse versteckt.

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



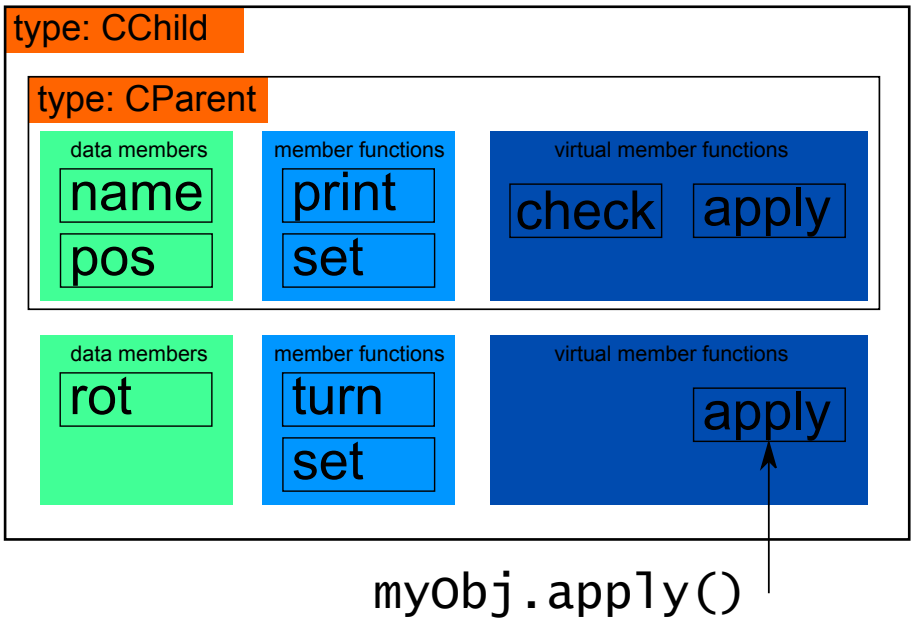
Zugriff auf member von myPrObj (3) - qualified-id

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

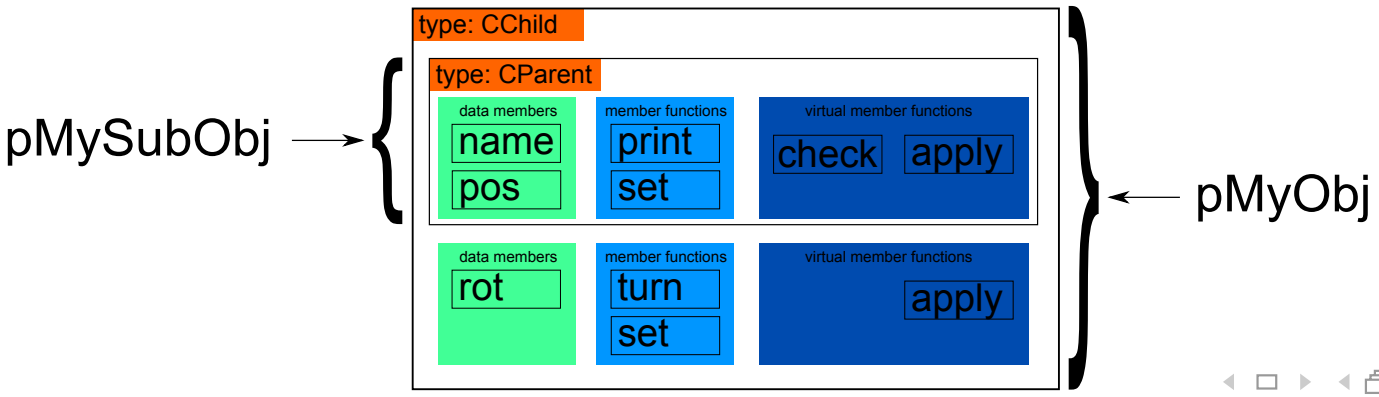
Wir können freilich einen Pointer auf `myObj` anlegen:

```
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

dynamic type (Standard, 1.3.3)

blabla lvalue blabla (unmittelbar unverständlich)

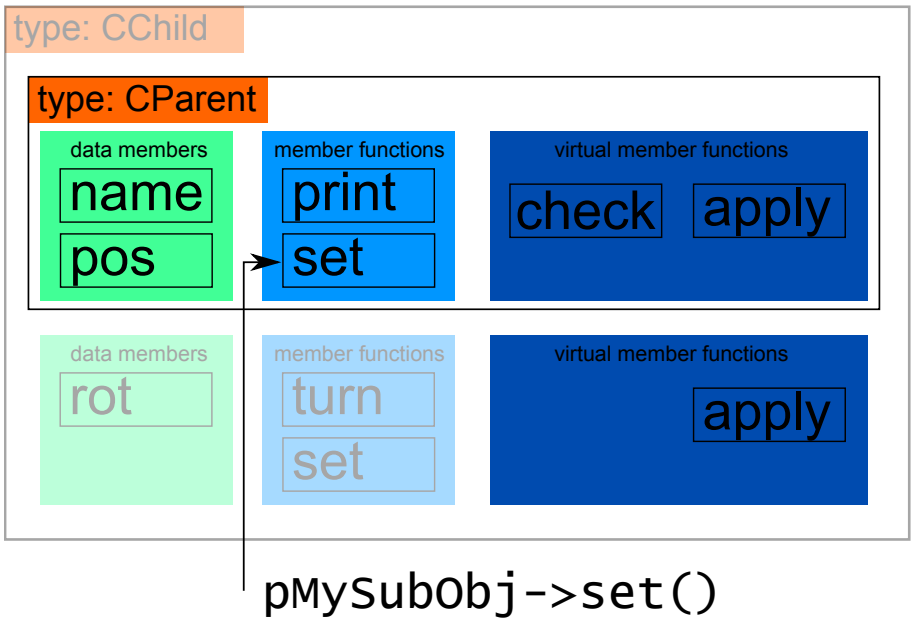
Beispiel:

`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)

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“.



2 Virtuelle Methoden

Überschreiben

Überschreiben von virtuellen Funktionen (Standard, 10.3:2)

In einer Klasse `CParent` sei eine virtual member function `apply` deklariert. Eine Klasse `CChild` sei direkt oder indirekt von `CParent` abgeleitet. 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 *überschreibt* `CParent::apply`. Der Einfachheit halber sagt man, dass Definition der virtual member function in der Elterklasse ebenfalls eine Überschreibung ist, wenn sie nicht als pure virtual member function angelegt wurde (*pure*: später).

Beachte: Überschreiben ist etwas Anderes als verdecken (name hiding)!

Das Aufrufen virtueller Funktionen (1)

Funktionsaufrufe (Standard, 5.2.2:1)

Wird eine virtual function aufgerufen, wird der dynamic type des ≫Dings≪ herangezogen. Es wird dann ausgehend vom dynamic type (meist-abgeleiteten Klasse) in der Vererbungshierarchie bzw. Stammbaum in Richtung der Elter-Klassen nach einer Überschreibung der Funktion gesucht und der erste Treffer verwendet.

Das Aufrufen virtueller Funktionen (1)

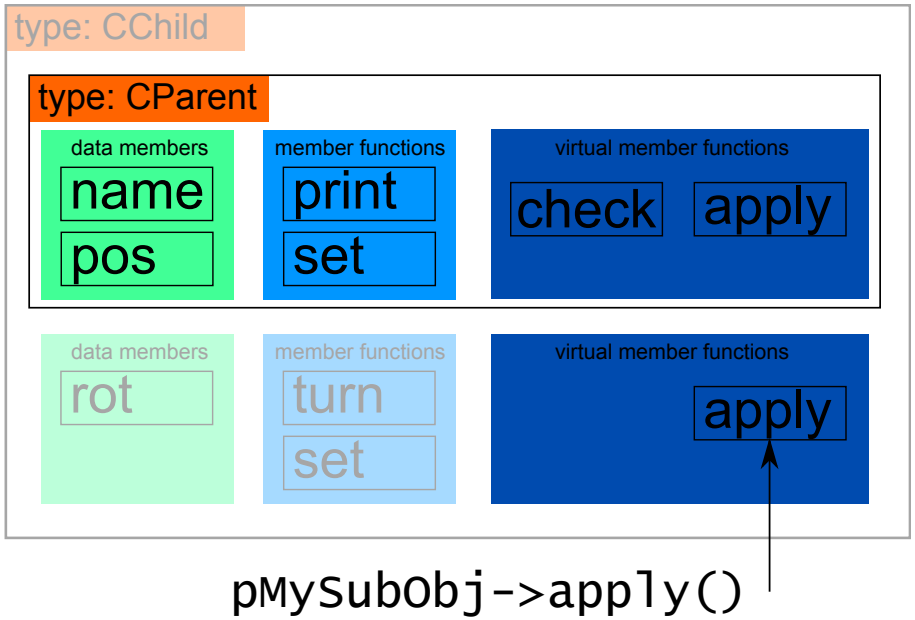
Funktionsaufrufe (Standard, 5.2.2:1)

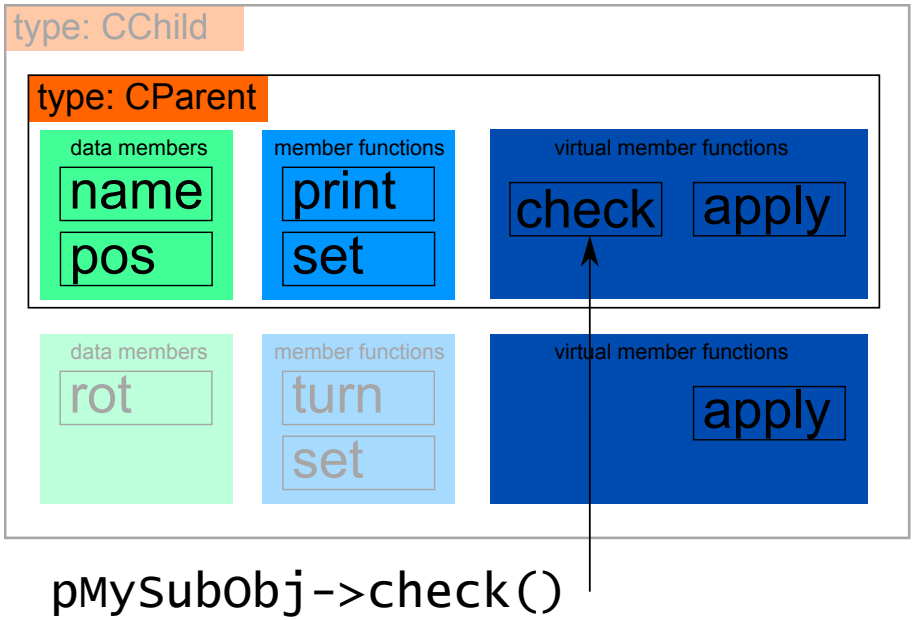
Wird eine virtual function aufgerufen, wird der dynamic type des ≫Dings≪ herangezogen. Es wird dann ausgehend vom dynamic type (meist-abgeleiteten Klasse) in der Vererbungshierarchie bzw. Stammbaum in Richtung der Elter-Klassen nach einer Überschreibung der Funktion gesucht und der erste Treffer verwendet.

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

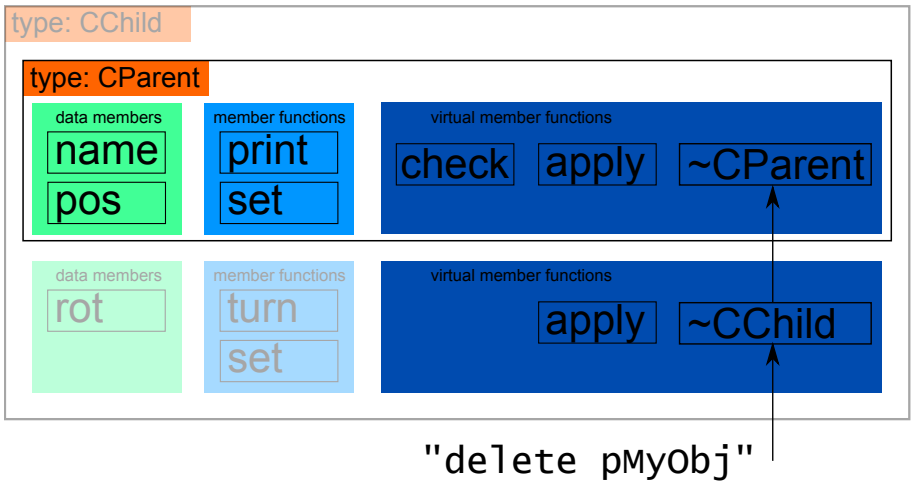
Angenommen, ich hätte `pMySubObj = new CChild;` geschrieben. Was macht dann `delete pMySubObj;`?

Der virtuelle Destruktor

Angenommen, ich hätte `pMySubObj = new CChild;` geschrieben. Was macht dann `delete pMySubObj;`?

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

- Ist der Destruktor *nicht* virtuell, so wird nur `pMySubObj->~CParent()` aufgerufen.
- Ist der Destruktor virtuell, so wird zunächst `pMySubObj->~CChild()` und anschließend `pMySubObj->~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* überschriebenen Destruktoren (die der Basisklassen) aufgerufen, und zwar beginnend mit dem dtor im meist abgeleiteten Typ, dann jeweils bei Ende des dtors einer Klasse der dtor der direkten Elterklasse dieser Klasse.

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* überschriebenen Destruktoren (die der Basisklassen) aufgerufen, und zwar beginnend mit dem dtor im meist abgeleiteten Typ, 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

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?

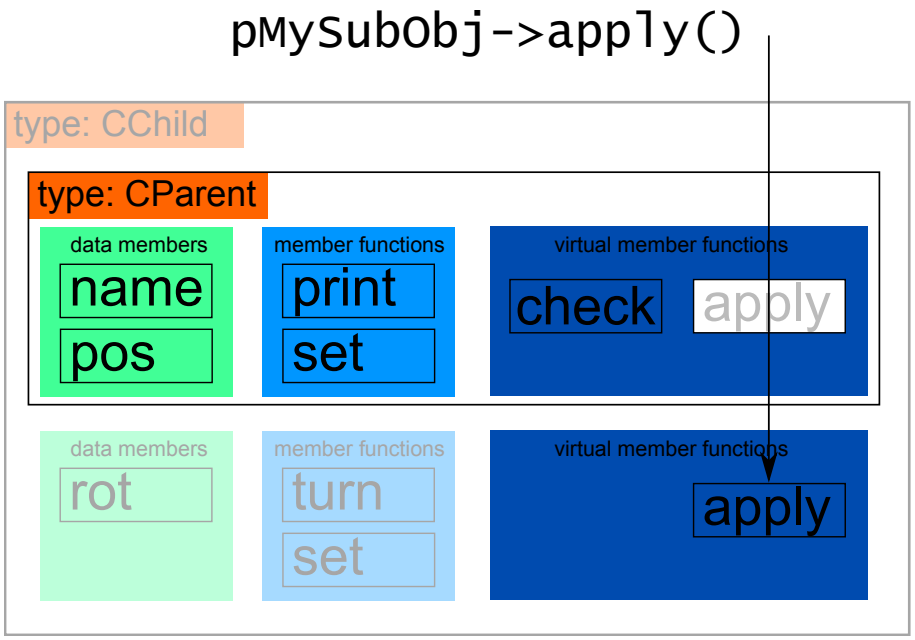
pure virtual member functions

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: Mit einem `BaseClass*`-Pointer kann man die pure virtual member function ansprechen. Was dann aufgerufen wird, ist eine Überschreibung in einer abgeleiteten Klasse.

pure virtual member functions: Beispiel

Machen wir zum Beispiel die Funktion `apply` in `CParent` zu einer pure virtual member function, also `virtual void apply() = 0;`



3 Abstrakte Klassen

Abstrakte Klassen

Es gibt einen Hauptunterschied zwischen: `virtual void apply() { }` und `virtual void apply() = 0;` Die erste Variante hat eine *leere* Implementierung (sie tut nichts), während die zweite Variante *keine* Implementierung hat (sie kann nicht selbst aufgerufen werden, sondern nur eine Überschreibung). Eine einzige solche pure virtual member function macht eine Klasse zu einer *abstrakten Klasse*.

Abstrakte Klassen

Es gibt einen Hauptunterschied zwischen: `virtual void apply() { }` und `virtual void apply() = 0;` Die erste Variante hat eine *leere* Implementierung (sie tut nichts), während die zweite Variante *keine* Implementierung hat (sie kann nicht selbst aufgerufen werden, sondern nur eine Überschreibung). Eine einzige solche pure virtual member function macht eine Klasse zu einer *abstrakten Klasse*.

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 keine Überschreibung in einer Elterklasse existiert).

Interface

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

Interface

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

Interfaces werden für sehr viele Dinge verwendet und sind neben dem Arbeiten mit Objekten (Instanzen) 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` abgeleitet ist und Überschreibungen aller pure virtual member functions enthält. Die letztere Seite erzeugt dann eine Instanz dieser abgeleiteten Klasse und gibt der ersten Seite einen `MyInterface*`-Pointer.

4 dynamic_cast

Der Vollständigkeit halber: dynamic_cast

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

Der Vollständigkeit halber: dynamic_cast

Man kann implizit von einem `CChild*` einen `CParent*` erhalten. Umgekehrt geht das nicht ohne weiteres, denn z.B. ein Pointer auf `myPrObj` kann nicht sinnvollerweise in einen Pointer auf eine `CChild`-Instanz 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` (z.B. `pMySubObj`), so ist das Ergebnis ein gültiger Pointer auf diese Instanz (auf das ≫Ding≪). Anderenfalls (z.B. `&myPrObj`) ist das Resultat `0`, ein ungültiger Pointer.