# Vererbung und RTTI

☑ **Lösung zu Aufgabe 6.1** Die Ausgaben sind den einzelnen Zeilen von folgendem Code zu entnehmen.

```
Child    c;       c.f();  c.g();  // Ausgabe: Child::f  Child::g
Self&    s =  c;  s.f();  s.g();  // Ausgabe: Child::f  Self::g
Parent   p =  c;  p.f();  p.g();  // Ausgabe: Parent::f Parent::g
Parent*  q = &c;  q->f(); q->g(); // Ausgabe: Child::f  Parent::g
```

Wegen dem Schlüsselwort `virtual` ist der Aufruf der Methode `f` immer abhängig vom dynamischen Typ (zur Laufzeit) des entsprechenden Objekts, während der Aufruf der Methode `g` vom statischen Typ (während der Kompilierzeit) der entsprechenden Variable abhängig ist. Dabei spielt es keine Rolle, ob wir mit einer Referenz (so wie in der zweiten Zeile) oder mit einem Zeiger (so wie in der vierten Zeile) auf das Objekt verweisen. Wir beobachten zudem, dass in der dritten Zeile eine neue Variable `p` (und damit ein neues Objekt) mit (statischem und dynamischem) Typ `Parent` erstellt wird, weshalb auch der Aufruf von `f` in dieser dritten Zeile zur (vielleicht etwas überraschenden) Ausgabe `Parent::f` führt.

☑ **Lösung zu Aufgabe 6.2** Die Ausgabe des Programms ist `A B C D`. Das heisst, dass alle bedingten Anweisungen ausgeführt werden ausser der letzten (die Ausgabe von `E` fehlt).

Allgemein ist zu sagen, dass sogenannte Upcasts von Zeigern in einer Vererbungshierarchie (so wie beispielsweise bei `A`) immer erlaubt sind. Eine solche Umwandlung funktioniert immer und geschieht deshalb sogar implizit ohne Verwendung von Schlüsselwörtern wie `static_cast` oder `dynamic_cast`.

Bei sogenannten Downcasts (so wie beispielsweise bei `B`, `C`, `D` und `E`) hingegen kann es im Prinzip vorkommen, dass der Zeiger zur Laufzeit des Programms auf ein Objekt zeigt, das gar nicht den gewünschten Typ hat. Deshalb müssen solche Umwandlungen explizit mit Schlüsselwörtern markiert werden.

Der Unterschied zwischen den Casts `static_cast` und `dynamic_cast` ist folgender: Der erste Cast wandelt zur Kompilierzeit den Typ des Zeigers wie gewünscht um; dabei wird ignoriert, ob diese Umwandlung zur Laufzeit des Programms tatsächlich Sinn macht. Der zweite Cast hingegen prüft zur Laufzeit des Programms, ob die Umwandlung sinnvoll ist; falls nicht, dann wird der Zeiger auf `nullptr` gesetzt.

Abschliessend ist zu bemerken, dass die Ausgabe `C` des Programms äusserst problematisch ist und auf einen Programmierfehler hindeutet. Aufgrund einer fehlerhaften Anwendung von `static_cast` wurde eine Zeigervariable `c` generiert, deren statischer Typ `Child*` nicht mit dem dynamischen Typ `Self` des Objekts, auf das gezeigt wird, im Einklang steht.

☑ **Lösung zu Aufgabe 6.3** Hier ist der gewünschte Code. Beachten Sie bitte, dass wir das Schlüsselwort `struct` nur deshalb anstelle von `class` einsetzen, um einige Zeilen Platz zu sparen.

```
class Animal {
protected:
  std::string m_name; // protected fuer Zugriff aus Cat, Dog, Bird
public:
  Animal(const std::string& name) : m_name(name) {}
  virtual ~Animal() = default; // virtual fuer dynamische Bindung
  virtual void say() = 0;      // = 0 fuer Abstraktheit der Klasse
};
struct Cat : public Animal {
  using Animal::Animal; // Konstruktor von Animal erben
  void say() override { std::cout << m_name << " says Meow! "; }
};
struct Dog : public Animal {
  using Animal::Animal; // Konstruktor von Animal erben
  void say() override { std::cout << m_name << " says Woof! "; }
};
struct Bird : public Animal {
  using Animal::Animal; // Konstruktor von Animal erben
  void say() override { std::cout << m_name << " says Chirp! "; }
};
```