

# Vererbung

Sebastian Stabinger

SS2020

# Das Problem

- Wir wollen Klassen für verschiedene Elemente eines Spiels deklarieren (z.B. Spieler, Ladenbesitzer, Objekt, ...)
- Viele Aspekte dieser Klassen sind gleich oder zumindest ähnlich. Alle haben eine aktuelle Position und ein Bild das angezeigt werden soll
- Ein paar Aspekte sind aber anders. z.B. hat ein Spieler evtl. Lebenspunkte und kann sich bewegen. Ein Ladenbesitzer hat ein Inventar das er verkaufen kann und ein Objekt kann passierbar sein oder nicht
- Mit unserem aktuellen Wissen müssten wir die gemeinsamen Eigenschaften in der Spieler-, Ladenbesitzer- und Objekt-Klasse manuell wiederholen.

# Beispiel Code — Spieler

```
class Player {  
private:  
    string _image;  
    int _xpos, _ypos, _health;  
  
public:  
    Player(int xpos, int ypos, int health, string image) {  
        _xpos = xpos;  
        _ypos = ypos;  
        _image = image;  
        _health = health;  
    }  
  
    void move_up() { ypos--; }  
    void move_down() { ypos++; }  
    void move_left() { xpos--; }  
    void move_right() { xpos++; }  
  
    bool isdead() { return health <= 0; }  
    void draw() { draw_img(_image, _xpos * 16, _ypos * 16); }  
};
```

# Beispiel Code — Ladenbesitzer

```
class Shopkeep {
private:
    string _image;
    int _xpos, _ypos;
    vector<string> _inventory;

public:
    Shopkeep(int xpos, int ypos, string image, vector<string> inventory) {
        _xpos = xpos;
        _ypos = ypos;
        _image = image;
        _inventory = inventory;
    }

    vector<string> get_inventory() { return _inventory; }

    void draw() { draw_img(_image, _xpos * 16, _ypos * 16); }
};
```

# Beispiel Code — Objekt

```
class Object {  
private:  
    int _xpos, _ypos;  
    string _image;  
    bool _solid;  
  
public:  
    Object(int xpos, int ypos, string image, bool solid) {  
        _xpos = xpos;  
        _ypos = ypos;  
        _image = image;  
        _solid = solid;  
    }  
  
    bool issolid() { return solid; }  
    void draw() { draw_img(_image, _xpos * 16, _ypos * 16); }  
};
```

# Die Lösung — Vererbung

# Vererbung

- Mittels Vererbung kann eine Klasse (die **abgeleitete Klasse**) Variablen und Memberfunktionen einer anderen Klasse (der **Basisklasse**) erben und somit verwenden.
- Wir erweitern und/oder ändern also die Funktionalität einer bereits existierenden Klasse (die **Basisklasse**) und geben dieser geänderten Klasse (die **abgeleitete Klasse**) einen neuen Namen.
- Wir sagen **die abgeleitete Klasse erweitert die Basisklasse**

## Syntax

```
class AbgeleiteteKlasse : public Basisklasse {  
    // ... Definition der abgeleiteten Klasse  
}
```

- **Achtung:** Die abgeleitete Klasse hat nur Zugriff auf Variablen und Memberfunktionen der Basisklasse welche **public** sind!



# Vererbung — Kurzes Beispiel

```
#include <iostream>
using namespace std;

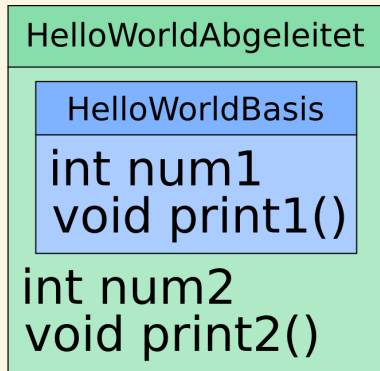
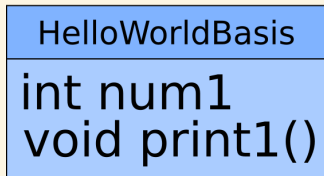
class HelloWorldBasis {
public:
    int num1 = 42;
    void print1() { cout << "Hallo: Basisklasse " << num1 << endl; }
};

class HelloWorldAbgeleitet : public HelloWorldBasis {
public:
    int num2 = 23;
    void print2() { cout << "Hallo: Abgeleitete Klasse " << num1 << endl; }
};

int main() {
    HelloWorldBasis w1;
    w1.print1(); // Output: Hallo : Basisklasse 42
    HelloWorldAbgeleitet w2;
    w2.print2(); // Output: Hallo : Abgeleitete Klasse 42
    // Die Funktion print1 ist auch in der abgeleiteten Klasse
    w2.print1(); // Output: Hallo : Basisklasse 42
    cout << "num = " << w2.num1 << endl; // 42
    cout << "num2 = " << w2.num2 << endl; // 23
}
```

# Vererbung — Kurzes Beispiel

Grafisch kann man sich die Vererbung folgendermaßen vorstellen  
(Die Basisklasse ist in die abgeleitete Klasse eingebettet):



# Überladen von Funktionen

# Funktionsweise

Wir können eine Funktion der Basisklasse in der abgeleiteten Klasse überschreiben indem wir sie gleich benennen.

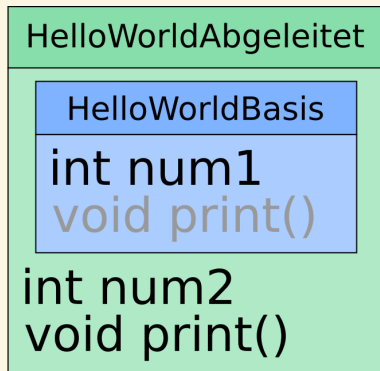
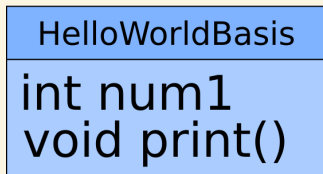
```
#include <iostream>
using namespace std;

class HelloWorldBasis {
public:
    int num1 = 42;
    void print() { cout << "Hallo: Basisklasse " << num1 << endl; }
};

class HelloWorldAbgeleitet : public HelloWorldBasis {
public:
    int num2 = 23;
    void print() { cout << "Hallo: Abgeleitete Klasse " << num1 << endl; }
};

int main() {
    HelloWorldBasis w1;
    w1.print(); // Output: Hallo : Basisklasse 42
    HelloWorldAbgeleitet w2;
    w2.print(); // Output: Hallo : Abgeleitete Klasse 42
    cout << "num = " << w2.num1 << endl; // 42
    cout << "num2 = " << w2.num2 << endl; // 23
}
```

# Überladen von Funktionen



- Das System sucht sozusagen von **Außen nach Innen** und führt die erste Funktion aus welche hinsichtlich Namen und Parametern passt

# Verwendung überschriebener Funktionen

## Problem

- Wir haben in `HelloWorldAbgeleitet` das vererbte `print` von `HelloWorldBasis` überschrieben
- Wir wollen irgendwo in `HelloWorldAbgeleitet` trotzdem das `print` von `HelloWorldBasis` aufrufen
- Was z.B. häufig vorkommt: Wir erweitern eine Funktion der Basisklasse indem wir sie überschreiben. In der neuen Funktion rufen wir die überschriebene Funktion der Basisklasse auf.

## Lösung

- Wir können auf die Basisklasse wie auf einen Namespace zugreifen.
- Bsp. `HelloWorldBasis::print()`

# Verwendung überschriebener Funktionen

```
#include <iostream>
using namespace std;

class HelloWorldBasis {
public:
    int num1 = 42;
    void print() { cout << "Hallo: Basisklasse " << num1 << endl; }
};

class HelloWorldAbgeleitet : public HelloWorldBasis {
public:
    int num2 = 23;
    void print() {
        HelloWorldBasis::print();
        cout << "Hallo: Abgeleitete Klasse " << num1 << endl;
    }
};

int main() {
    HelloWorldAbgeleitet w2;
    w2.print();
    // Output:
    // Hallo : Basisklasse 42
    // Hallo : Abgeleitete Klasse 42
}
```

# Konstruktor und Vererbung



# Standardkonstruktoren

Der Standardkonstruktor einer Basisklasse wird automatisch aufgerufen wenn die abgeleiteten Klasse erzeugt wird.

```
#include <iostream>
```

```
class C1 {  
public:  
    int i;  
    C1() { i = 23; }  
};
```

```
class C2 : public C1 {  
public:  
    int j;  
    C2() { j = 42; }  
};
```

```
int main() {  
    C2 c;  
    std::cout << c.i << " " << c.j << std::endl; // Output: 23 42  
}
```

# Konstrukoren mit Parametern

Ein Konstruktor der Basisklasse welcher Parameter erwartet **muss explizit aufgerufen werden**. Dies geschieht durch Anhängen mittels **:** an den Konstruktor.

## Syntax — Beispiel

```
class C1 {
public:
    int a;
    C1(int pa) { a = 2 * pa; }
};

class C2 : public C1 {
public:
    C2(int i) : C1(3 * i) { a = a + 10; }
};

int main() {
    C2 c(4); // c.a == 34 [(2*(3*4))+10]
    cout << c.a << endl;
}
```

# Ein größeres Beispiel

# Code Beispiel — Ein Ding

```
class Thing {  
private:  
    int _xpos, _ypos;  
    string _image;  
  
public:  
    Thing(int xpos, int ypos, string image) {  
        _xpos = xpos;  
        _ypos = ypos;  
        _image = image;  
    }  
  
    int get_xpos() { return _xpos; }  
    int get_ypos() { return _ypos; }  
    void set_xpos(int x) { _xpos = x; }  
    void set_ypos(int y) { _ypos = y; }  
  
    void draw() { draw_image(_image, _xpos * 16, _ypos * 16); }  
};
```

# Code Beispiel — Spieler

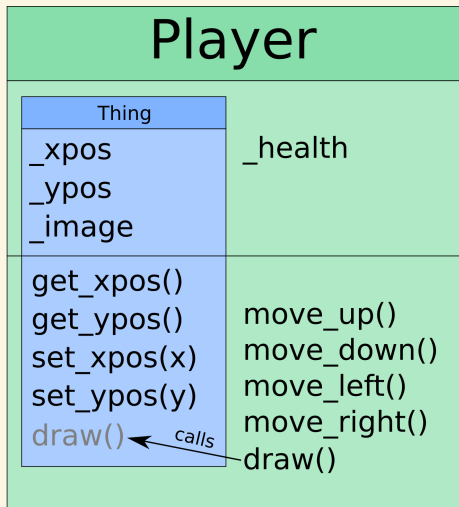
```
class Player : public Thing {
private:
    int _health;

public:
    Player(int xpos, int ypos, int health, string image)
        : Thing(xpos, ypos, image) {
        _health = health;
    }

    void move_up() { set_ypos(get_ypos() - 1); }
    void move_down() { set_ypos(get_ypos() + 1); }
    void move_left() { set_xpos(get_xpos() - 1); }
    void move_right() { set_xpos(get_xpos() + 1); }

    void draw() { // Erweitere draw Funktion von Thing
        Thing::draw();
        // Zeichne Gesundheitsanzeige ...
    }
};
```

Thing
<code>_xpos</code> <code>_ypos</code> <code>_image</code>
<code>get_xpos()</code> <code>get_ypos()</code> <code>set_xpos(x)</code> <code>set_ypos(y)</code> <code>draw()</code>



```
class Shopkeep : public Thing {  
private:  
    vector<string> _inventory;  
  
public:  
    Shopkeep(int xpos, int ypos, string image, vector<string> inventory)  
        : Thing(xpos, ypos, image) {  
        _inventory = inventory;  
    }  
  
    vector<string> get_inventory() { return _inventory; }  
};
```

# Code Beispiel — Objekt

```
class Object : public Thing {  
private:  
    bool _solid;  
  
public:  
    Object(int xpos, int ypos, string image, bool solid)  
        : Thing(xpos, ypos, image) {  
        _solid = solid;  
    }  
  
    bool issolid() { return _solid; }  
};
```



# Kontrolle der Sichtbarkeit

# Das Problem

- Wir haben bereits gelernt: Die abgeleitete Klasse hat Zugriff auf alle Elemente der Basisklasse welche **public** sind. Auf die **private** Elemente hat sie **keinen** Zugriff.
- Wir wollen aber häufig in der abgeleiteten Klasse direkt auf Elemente der Basisklasse zugreifen welche von außerhalb nicht sichtbar sind.

## Beispiel

```
class C1 {  
private:  
    int secret;  
};  
  
class C2 : public C1 {  
public:  
    void change_secret(int newsecret) { secret = newsecret; }  
    // Error!! Kein Zugriff auf secret  
};
```

Wir könnten **secret** in den **public** Bereich schieben, aber dann kann jeder darauf zugreifen ...

# Die Lösung: protected

- Um dieses Problem zu lösen gibt es einen dritten Sichtbarkeitsbereich innerhalb einer Klasse namens **protected**
- Er verhält sich von Außerhalb so wie **private**, aber eine abgeleitete Klasse kann direkt darauf zugreifen.

## Beispiel

```
class C1 {
protected:
    int secret;
};

class C2 : public C1 {
public:
    void change_secret(int newsecret) { secret = newsecret; }
};

int main() {
    C2 c;
    c.change_secret(42); // Alles OK
    c.secret;           // Error!! Kein Zugriff auf secret
}
```

# Übungen

Wir schreiben die beim letzten Termin entwickelte Version des Spiels so um, dass es eine **Spieler-** und eine **Monsterklasse** gibt.

- Der Spieler bekommt eine Gesundheitsanzeige
- Die Monster verschwinden wie gehabt nach einer gewissen Zeit, wir lösen das dieses mal aber in der Monsterklasse selbst