Einführung in die Programmierung mit C++

Klassenhierarchien

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

Klassenhierarchien

Motivation


```
1  class live_countable {
2     static size_t c;
3  public:
4     live_countable() { c++; }
5     ~live_countable() { c--; }
6     static const size_t& get_live_count() {
7         return c;
8     }
9  };
10
11 size_t live_countable::c=0;
12
13 template<typename F, typename S>
14 class pair : public live_countable {
15    ...
16 }
```

- Aufteilung von Funktionalitäten auf Klassenhierarchien im Sinne von is-a Relationen wird unterstützt, z.B. allozierte Objekte vom Typ pair sollten zählbar (vom Typ live_countable) sein.
- Zugriffsrechte regulieren Zugang zu Variablen und Funktionen der Generalisierungen (z.B. live_countable) innerhalb der Spezialisierungen (z.B. pair).

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

Klassenhierarchien

Vererbung


```
1  #include<iostream>
2
3  struct B { int i=3; };
4
5  struct D1 : public B {
6    int j=5;
7  };
8
9  struct D2 : public D1 {
10   int k=7;
11 };
12
13 int main() {
14   using namespace std;
15   B b; cout << b.i << endl;
16   D1 d1; cout << d1.i*d1.j << endl;
17   D2 d2; cout << d2.i*d2.j*d2.k << endl;
18   return 0;
19 }
```

→ Tafel: UML Notation

- Unterklassen (abgeleitete Klassen) erben die nichtprivaten Daten und Funktionen ihrer Oberklassen; z.B. erbt D1 die Variable i aus B; D2 erbt die Variablen i und j aus D1.
- Z.B. berechnet nebenstehendes Programm die Resultate der (eindeutigen) Primzahlfaktorisierungen von 3, 15 und 105, d.h. die Ausgabe

  3
  15
  105

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

Klassenhierarchien

Zugriffsrechte


```
1  #include<iostream>
2  using namespace std; // :-( ...
3
4  class B {
5    public: int i_publ=1;
6    protected: int i_prot=2;
7    private: int i_priv=3;
8  };
9
10 class D : public B { public:
11   void print() {
12     cout << i_publ << i_prot << endl;
13     // cout << i_priv << endl;
14   }
15 };
16
17 int main() {
18   D d; d.print(); cout << d.i_publ << endl;
19   // cout << d.i_prot << endl;
20   // cout << d.i_priv << endl;
21   return 0;
22 }
```

- Variablen und Funktionen, welche an Unterklassen vererbt werden jedoch vor externem Zugriff geschützt sein sollen, werden als **protected** deklariert.
- Z.B. würden die auskommentierten Zeilen in nebenstehendem Programm Übersetzungsfehler nach sich ziehen.
- Übrigens ist ein globales **using namespace** std suboptimal.

→ Tafel: UML Notation

- In

  ```
  class D : public B { ... };
  ```

  erbt D alle als **public** oder **protected** deklarierten Daten und Funktionen aus B. Alle **public** Daten und Funktionen aus B sind ebenfalls **public** in D. Alle **protected** Daten und Funktionen bleiben **protected**.

- In

  ```
  class D : protected B { ... };
  ```

  werden alle als **public** deklarierten Daten und Funktionen aus B **protected** in D.

- In

  ```
  class D : private B { ... }; // equiv. class D : B { ... };
  ```

  werden alle als **public** oder **protected** deklarierten Daten und Funktionen aus B **private** in D.

→ Tafel: UML Notation

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

```
1  #include <iostream>
2  using namespace std;
3
4  class B {
5  public:
6    B() { cout << "B()" << endl; }
7    ~B() { cout << "~B()" << endl; }
8  };
9
10 class D1 : public B {
11 public:
12   D1() { cout << "D1()" << endl; }
13   ~D1() { cout << "~D1()" << endl; }
14 };
15
16 class D2 : public D1 {
17 public:
18   D2() { cout << "D2()" << endl; }
19   ~D2() { cout << "~D2()" << endl; }
20 };
21
22 int main() { D2 c; }
```

- Konstruktion *bottom-up*
- Destruktion *top-down*
- Z.B. generiert nebenstehendes Programm die folgende Ausgabe

```
B()
D1()
D2()
~D2()
~D1()
~B()
```

AUFWACHEN!

Kann folgendes Programm erfolgreich übersetzt werden?

```
1  class B {
2    protected: int i;
3  };
4
5  class D1 : public B {
6    int j; int foo() { return i; }
7  };
8
9  class D2 : protected D1 {
10   protected: int k; int foo() { return i; }
11 };
12
13 int main() {
14   D1 a; a.foo(); int i=a.i;
15   D2 b; b.foo(); i=b.i;
16 }
```

Wach?

```
1  class B {
2    protected: int i;
3  };
4
5  class D1 : public B {
6    int j; int foo() { return i; }
7  };
8
9  class D2 : protected D1 {
10   protected: int k; int foo() { return i; }
11 };
12
13 int main() {
14   D1 a; a.foo(); int i=a.i;
15   D2 b; b.foo(); i=b.i;
16 }
```

foo **private** in D1, i **protected** in D1, foo **protected** in D2 und i **protected** in D2.

WACH BLEIBEN!

Wer muss wessen Freund (**friend**) werden, damit dasselbe Programm erfolgreich übersetzt werden kann?

Noch Wach?

```
1  class B {
2    protected: int i;
3  };
4
5  class D1 : public B {
6    int j; int foo() { return i; }
7    friend int main();
8  };
9
10 class D2 : protected D1 {
11   protected: int k; int foo() { return i; }
12   friend int main();
13 };
14
15 int main() {
16   D1 a; a.foo(); int i=a.i;
17   D2 b; b.foo(); i=b.i;
18 }
```

```
1  #include<iostream>
2
3  class B {
4    public:
5      int i;
6      B(int i) : i(i) {}
7  };
8
9  class D : public B {
10   public:
11     int j;
12     D(int j) : B(j-2), j(j) {}
13 };
14
15 int main() {
16   D d(4);
17   std::cout << d.j << d.i << std::endl;
18   return 0;
19 }
```

- Konstruktoren der Basis können bei Konstruktion der Spezialisierung innerhalb der Initialisierungliste explizit aufgerufen werden.
- Alternativ kann (sofern der Standardkonstruktor der Basis definiert ist; hier nicht) auch auf die implizite Standardkonstruktion gebaut und den Variablen explizit Werte zugewiesen werden.

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

- Dominanz von Daten:

```
1 class B { public: int a; };
2 class D : public B { public: float a; };
```

D erbt a von B, welches jedoch durch die lokale Deklaration von a in D dominiert wird.

- Zugriff auf das ganzzahlige a in D y geschieht über y.B::a.

- Überschreiben von Funktionen

```
1 class B { public: void f(); };
2 class D : public B { public: void f(); };
```

D erbt f von B, welches jedoch durch die lokale Deklaration von f in D überschrieben wird.

- Zugriff auf die aus B stammende Funktion f in D y geschieht über y.B::f().

AUFWACHEN!

Was wird durch das folgende Programm ausgegeben?

```
1  #include<iostream>
2
3  class B {
4    protected: int i;
5    public: int get() { return i; }
6  };
7
8  class D : public B {
9    float i;
10   public: D(int j,float f): i(f) { B::i=j;};
11             float get() { return i; }
12 };
13
14 int main() {
15   using namespace std;
16   D d(1,2.2);
17   cout << d.get() << d.B::get() << endl;
18   return 0;
19 }
```

Wach?

```
1  #include<iostream>
2
3  class B {
4     protected: int i;
5     public: int get() { return i; }
6  };
7
8  class D : public B {
9     float i;
10    public: D(int j,float f): i(f) { B::i=j;};
11              float get() { return i; }
12 };
13
14 int main() {
15    using namespace std;
16    D d(1,2.2);
17    cout << d.get() << d.B::get() << endl;
18    return 0;
19 }
```

2.21

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

Spezialisierungen können von mehreren Basisklassen abgeleitet werden, z.B.

```
1 class D1 {}; class B {};
2 class D2 : public D1, public B {};
```

Probleme können z.B. bei Abhängigkeitein in “Diamantenform” enstehen.

```
1 class B { void f(); };
2 class D11 : public B {};
3 class D12 : public B {};
4 class D2 : public D11, public D12 {};
```

Welches f() soll in D2 verwendet werden?

→ [**using**] D11::f() oder [**using**] D12::f() in D2.

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

```
1  struct B {
2    int i=1;
3  };
4
5  struct D : public B {
6    int j=2;
7  };
8
9  int main () {
10   D d;
11   B* bp=&d; B& br=d; // ok
12   bp->i+=br.i; // ok
13   bp->j+=br.j; // error
14   return 0;
15 }
```

- Zeiger und Referenzen können jeweils polymorph (von verschiedender Art) sein.
- Objekte von Spezialisierungen sind erweiterte Instanzen der Basis. Sie teilen sich dieselbe Basisadresse.
- Zeiger auf die Basis können auch auf Objekte der Spezialisierung zeigen.
- Rerenzen für die Basis können auch auf Objekte der Spezialisierung referenzieren.
- Zugriff auf die Daten der Basis funktioniert wie gehabt. Für den Zugriff auf Daten der Spezialisierung werden virtuelle Funktionen benötigt.

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

```
1  struct B {
2    int i=1;
3    virtual int& ref() { return i; }
4  };
5
6  struct D : public B {
7    int j=2;
8    int& ref() { return j; }
9  };
10
11 int main () {
12   D d;
13   B* bp=&d; B& br=d; // ok
14   bp->ref()+=br.ref(); // ok
15   bp->B::ref()+=br.B::ref(); // ok
16   return 0;
17 }
```

- Wird eine virtuelle Funktion via einen polymorphen Zeiger (bzw. Referenz) aufgerufen, so wird die Implementierung der Funktion in der Spezialsierung ausgeführt, z.B.
  `bp->ref()+=br.ref();`
- Virtuelle Funktionen sollten auf allen Stufen einer Klassenhierarchie implementiert werden. Es wird die Implementierung der höchsten Spezialisierung ausgeführt, für die eine solche Implementierung existiert, siehe nächste Folie.

```
1  struct B {
2    int i=1;
3    virtual int& ref() { return i; }
4  };
5
6  struct D1 : public B {
7    int j=2;
8    int& ref() { return j; }
9  };
10
11 struct D2 : public D1 { int k=3; };
12
13 int main () {
14   D2 d; B* bp=&d; B& br=d; // ok
15   bp->ref()+=br.ref(); // ok -> j=4
16   return 0;
17 }
```

- Die höchste Spezialisierung mit existierender Implementierung von ref stellt D1 dar.
- Der Wert von j wird verdoppelt.
- Die Variable k ist nicht erreichbar.

AUFWACHEN!

Was wird durch folgendes Programm ausgegeben?

```
1  #include<iostream>
2
3  class B { public:
4    virtual int type() { return 0; }
5  };
6
7  class D : public B { public:
8    int type() { return 1; }
9  };
10
11 int main() {
12   B b; D d, *bp2=&d; B& br=*bp2;
13   std::cout << b.type() << bp2->type()
14         << br.type() << std::endl;
15   return 0;
16 }
```

Wach?

```
1  #include<iostream>
2
3  class B { public:
4    virtual int type() { return 0; }
5  };
6
7  class D : public B { public:
8    int type() { return 1; }
9  };
10
11 int main() {
12   B b; D d, *bp2=&d; B& br=*bp2;
13   std::cout << b.type() << bp2->type()
14         << br.type() << std::endl;
15   return 0;
16 }
```

011

Outline

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen

```
1  #include<iostream>
2
3  class B { public:
4    virtual int type()=0;
5  };
6
7  class D : public B { public:
8    int type() { return 1; }
9  };
10
11 int main() {
12   B b; // error
13   D d; // ok
14   B *dp=&d, &dr=d; // ok
15   std::cout << d.type() << dp->type()
16       << dr.type() << std::endl;
17   return 0;
18 }
```

- Klassen mit mindestens einer rein virtuellen Funktion sind abstrakt.
- Abstrakte Klassen dürfen nicht instanziert werden. Sie definieren Mindestanforderungen an die Schnittstelle abgeleiteter Spezialisierungen.

AUFWACHEN!

Was wird durch folgendes Programm ausgegeben?

```
1  #include<iostream>
2
3  class B {
4    public:
5      virtual int type()=0;
6  };
7
8  class D : public B {
9    public:
10     int type() { return 1; }
11 };
12
13 int main() {
14   D d;
15   B* vp[2]={&d,vp[0]};
16   std::cout << (vp[0]<=vp[1]) << std::endl;
17   return 0;
18 }
```

Wach?

```
1  #include<iostream>
2
3  class B {
4    public:
5      virtual int type()=0;
6  };
7
8  class D : public B {
9    public:
10     int type() { return 1; }
11 };
12
13 int main() {
14   D d;
15   B* vp[2]={&d,vp[0]};
16   std::cout << (vp[0]<=vp[1]) << std::endl;
17   return 0;
18 }
```

1

Zusammenfassung

Motivation

Vererbung

Zugriffsrechte

Konstruktion und Destruktion

Dominanz und Überschreiben

Mehrfachvererbung

Polymorphie

Virtuelle Funktionen

Abstrakte Klassen