Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

- inline-Funktionen
- Überladen von Operatoren

Tafelproblem der Woche: Der [][]-Operator

Operator Obfuscation

- Here is an example of elevating overloading operator obfuscation to a high art:
  - Overload the '`!`' operator for a class, but have the overload have nothing to do with inverting or negating
  - Make it return an integer
  - Then, in order to get a logical value for it, you must use '`! !`'
  - However, this inverts the logic
  - So [*drum roll]* you must use '`! ! !`'
- Another hint: Confuse the `!` operator, which returns a boolean 0 or 1, with the ~ bitwise logical negation operator

Gliederung

C++ Klassen und Vererbung (1)

- Beispiel:

```
class Fruit {
  string color;
  int seeds;
public:
  Fruit() : color(""), seeds(0) { }
  Fruit(string col, int s) : color(col), seeds(s) { }
  int getSeeds() const { return seeds; }
  string getColor() const { return color; }
  void print() const {
    cout<< "I am a Fruit!" << endl;
  }
};
```

- Nun erstellen wir verschiedene Instanzen

```
Fruit pear("green", 10);       // pear is a Fruit
Fruit orange("orange", 25);    // orange is a Fruit
```

C++ Klassen und Vererbung (2)

- Der Aufruf der Ausgabe

```
pear.print();
orange.print();
```

- ergibt

```
I am a Fruit!
I am a Fruit!
```

- Nun möchten wir die Eigenschaft repräsentieren, dass die Orange geschält werden kann
- Leider kann `Fruit` das nicht repräsentieren
- Und pears lassen sich nicht schälen
- Lösung: Eine Klasse `Orange`, die die Klasse `Fruit` erweitert!

C++ Klassen und Vererbung (3)

- Die Klasse Orange:

```
class Orange : public Fruit {
  bool peeled; // True if peel has been removed
public:
  Orange() : peeled(false) { }
  bool isPeeled() const { return peeled; }
  void peel() {
    if (peeled) cout << "Already peeled!");
    peeled = true;
  }
};
```

- Orange besitzt zu der Funktionalität der Klasse Fruit neue Funktionalität
- Man kann auch `private` ableiten!

C++ Klassen und Vererbung (4)

- Erweiterung, weil
  - Eine Orange eine Frucht ist
  - Eine Orange die Charakteristik einer Frucht hat
  - Mehr Merkmale, Zustände und Funktionalität hinzugefügt werden
  - Eine Orange eine spezialisierte Version einer Frucht ist

- Neue Terminologie:
  - Fruit ist eine Oberklasse / Super Class / Base Class / Parent Class
  - Orange ist eine Unterklasse / abgeleitete Klasse / Derived Class / Child Class

C++ Klassen und Vererbung (5)

- Anwendungsbeispiel

```
Orange o;
cout << "This orange has " << o.getSeeds()
     << "seeds." << endl;
if (!o.isPeeled()) {
  cout << "Commencing peel removal..." << endl;
  o.peel();    // Peel the orange.
}
```

- Überall, wo wir `Fruit` benutzt haben, können wir jetzt auch `Orange` benutzen!

```
Fruit *pf1 = new Fruit();    // OK
Fruit *pf2 = new Orange();   // Also OK
```

- Eine `Orange` ist also eine `Fruit`
- Eine `Orange` hat die Member, die auch `Fruit` hat

C++ Klassen und Vererbung (6)

- Die Umkehrung gilt nicht!

```
Orange *po1 = new Orange(); // OK
Orange *po2 = new Fruit();  // Compiler error!
```

- Eine `Fruit` ist keine `Orange`
- `Fruit` hat nicht die Member einer `Orange`
- Man kann eine `Fruit` nicht schälen
- Die `Fruit` wurde wie folgt deklariert:

```
class Fruit {
  string color;
  int seeds;  ...
```

- Kann eine Orange auf die Variable `seeds` zugreifen (private)?

```
void Orange::removeSeeds() {
  seeds = 0;
}
```

C++ Klassen und Vererbung (7)

```
void Orange::removeSeeds() {
  seeds = 0;
}
```

- Nein.
- Nur die Klasse selbst kann auf private Elemente zugreifen
- Um Member in Unterklassen zugreifbar zu machen, müssen sie als `protected` definiert sein:

```
class Fruit {
protected: // Make accessible to subclasses!
  string color;
  int seeds;
  ...
}
```

- Nun funktioniert obiges Beispiel

C++ Klassen und Vererbung (8)

- Was sollte `private` sein?
- Was sollte `protected` sein?
- Keine Regel hier, nur Hinweise:
  - In der Oberklasse sollten die Member `private` sein, bis man in einer abgeleiteten Klasse bemerkt, dass man direkt zugreifen muss. Dann sollte man sie auf `protected` abändern.
  - Bei komplizierten Datenstrukturen sollten eher Zugriffsfunktionen verwendet werden, da die abgeleitete Klasse ansonsten viel durcheinander bringen kann.
- Eine abgeleitete Klasse kann Funktionen der Basisklasse überschreiben
- Funktionalität wird ersetzt / spezialisiert

C++ Klassen und Vererbung (9)

- Beispiel:

```
class Orange : public Fruit {
  ...
  void print() const { // Override Fruit::print()
    cout << "I am an Orange!" << endl;
  }
};
```

- Nun ergibt der Aufruf von `print()` einer Instanz von `Orange` folgendes:

```
I am an Orange!
```

C++ Klassen und Vererbung (10)

- Aufruf der `print()`-Funktion der Oberklasse

```
class Orange : public Fruit {
  ...
  void print() const {
    cout << "I am an Orange!" << endl;
    // Call parent-class version of print()
    Fruit::print();
  }
};
```

- Beispiel:

```
Fruit f;
Orange o;

f.print();
o.print();
```

führt zu:

```
I am a Fruit!
I am an Orange!
I am a Fruit!
```

C++ Klassen und Vererbung (11)

- Wie sieht es mit folgendem Code-Fragment aus?

```
void printFruit(const Fruit &fr) {
  fr.print();
}
...
Fruit f;
Orange o;
printFruit(f);
printFruit(o); // An orange is a fruit.
```

- Die Ausgaben sind:

```
I am a Fruit!
I am a Fruit!
```

- Standardmäßig benutzt C++ den Variablentyp, um zu identifizieren, welche Funktion aufgerufen wird
- Nicht den Objekttyp!

C++ Klassen und Vererbung (12)

```
void printFruit(const Fruit &fr) {
  fr.print();
}
```

- `fr` ist vom Type `Fruit`, also wird `Fruit::print()` aufgerufen
- Sogar wenn `fr` eine `Orange` ist!

```
Orange o;
printFruit(o);
```

- `virtual` sagt dem Compiler, dass er prüfen soll, ob die Funktion einer Unterklasse aufgerufen werden muss

C++ Klassen und Vererbung (13)

- Aktualisierung der Klasse Fruit:

```
class Fruit {
  ...
  virtual void print() const {
    cout << "I am a Fruit!" << endl;
  }
};
```

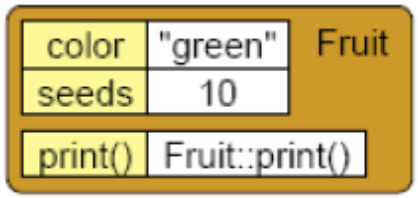
- Dann gibt es die Ausgaben

```
I am a Fruit!
I am an Orange!
```

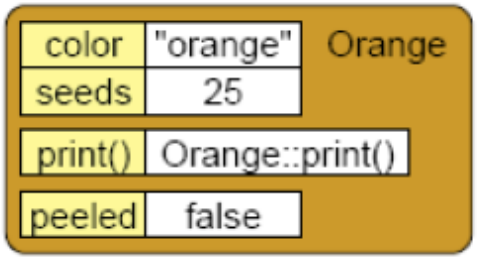
- Durch die Deklaration einer Methode als `virtual` wird der C++-Compiler gezwungen, einen Pointer zu der Instanzfunktion zu speichern
- Damit weiß das Objekt selbst, welche Methode aufgerufen werden muss

C++ Klassen und Vererbung (14)

- Wenn eine `Fruit` erstellt wird, wird `Fruit::print()` als Pointer gespeichert:



- Wenn eine Orange erstellt wird, wird `Orange::print()` gespeichert:



- „Virtual Function Table“
- Benötigte Methoden werden erst zur Laufzeit bestimmt
- Overhead durch Lookup und Dereferenzierung der Funktion-Pointer

C++ Klassen und Vererbung (15)

- Nicht-virtuelle Member können zur Compilezeit bestimmt werden
- Schneller
- Also: Sparsam mit `virtual` sein!
- Wie sieht das mit den Destruktoren aus?

```
Fruit *pf1 = new Fruit();
Fruit *pf2 = new Orange();
...
delete pf1; // Clean up my fruit.
delete pf2;
```

- Beide Varianten rufen `Fruit::~Fruit()` auf!

C++ Klassen und Vererbung (16)

- Der C++-Standard besagt, dass das Löschen einer abgeleiteten Klasse mittels eines Pointers der Basisklasse in einem *undefinierten* Verhalten resultiert!
- Lösung:
- Jede Basisklasse benötigt einen virtuellen Destruktor
- Dann wird schon der richtige Destruktor aufgerufen...

Abstrakte Klassen (1)

- Manche Basisklassen definieren nur Verhalten, sonst nichts
- Die Konzepte einer solchen Basisklasse sind zu allgemein, als dass sie bereits implementiert werden können
- Abstrakte Basisklassen müssen erweitert werden
- Abstrakte Klassen kann man nicht instanzieren
- Implementation der geforderten Funktionalität wird in den Unterklassen definiert
- Eine Klasse ist abstrakt, sobald sie mindestens eine Methode als „pure virtual“ deklariert.

Abstrakte Klassen (2)

- Beispiel:

```
class Figure {
  // ...
public:
  virtual void draw() const {}
};
```

- Besser:

```
class Figure {
  // ...
public:
  virtual void draw() const = 0;
};
```

- Nun müssen wir für jede Figure bestimmen, wie sie gezeichnet wird:

Abstrakte Klassen (3)

```
class Flowchart : public Figure {
    virtual void draw() {
        // Code to draw contends
    }
}
```

- Der Versuch eine `Figure` zu instanzieren führt zu einem Compilerfehler

```
Figure *fig = new Figure();     // Compiler Error
```

- Dennoch kann man `Figure`-Variablen haben:

```
Figure *fig = new Flowchart(); // OK
```

- Basisklassen können einige rein virtuelle Methoden haben

Abstrakte Klassen (4)

- Basisklassen können einige virtuelle Methoden haben
  - Basisklasse stellt dennoch einiges an Implementierungen zur Verfügung
  - Die abgeleitete Klasse ergänzt die fehlenden Teile
- Ein „Interface“ ist eine Basisklasse, die ausschließlich virtuelle Methoden enthält
  - Keine Implementation
  - Nur Verhaltensdefinition
  - Dieses Interface-Konzept ist ausgeprägter in anderen Sprachen

Basisklassen-Konstruktoren (1)

- Unsere Basisklassen `Fruit` hatte einen Konstruktor:

```
Fruit(string col, int s) : color(col), seeds(s) { }
```

- Wir wollen der abgeleiteten Klassen Orange ähnliches Verhalten geben:

```
Orange(string col, int s, bool p) :
  color(col), seeds(s), peeled(p) { }
```

- Funktioniert so nicht!
- Die abgeleitete Klassen darf die Member der Basisklasse nicht in MILs initialisieren
- Man muss den Fruit-Kontruktor in der MIL aufrufen:

```
Orange(string col, int s, bool p) :
  Fruit(col, s), peeled(p) { }
```

Basisklassen Konstruktoren (2)

- Manchmal hat eine Basisklasse keinen default-Konstruktor (Konstruktor ohne Argumente)
- Dann gibt es eine Fehlermeldung, wenn man ableitet, ohne einen geeigneten Konstruktor der Oberklasse aufzurufen
- In diesem Fall muss die abgeleitete Klasse den Kontruktor der Basisklasse in der MIL aufrufen!
- Wenn ein Konstruktor der Basisklasse in der Initializer-Liste aufgerufen wird, unbedingt an erster Stelle!!!

Objektorientiertes Programmieren in C++

- Wir hatten:
- Verkapselung: Verstecken der Details der Implementierung
- Abstraktion: Präsentation eines einfachen high-level Interfaces
- Vererben: Abgeleitete Klassen erben das Verhalten Ihrer Vorgänger
- Polymorphismus: Der Aufruf einer Methode kann basierend auf dem Objekttyp zu verschiedenen Verhalten führen
- In C++ geht das mittels virtual-Funktionen
- Man kann Interfaces realisieren, indem man Klassen baut, die ausschließlich rein virtuelle Methoden haben
- Man kann in C++ noch mehr machen
- Später im Teil „C++ für Fortgeschrittene“

Gliederung

C++ Streams (1)

- C++ stellt ein allgemeines Konzept für die Ein- und Ausgabe zur Verfügung
- Datenströme in C++:
  - Output: Konvertierung von Variablen und Objekten in Zeichensequenzen
  - Input: Konvertierung von Zeichensequenzen in Dateien in Variablen und Objekte
  - Formatierung spielt eine Rolle und wird definiert / programmiert
- Standard-Klassen unterstützen die Standard-Streams
- C++-Streams können leicht an eigene Typen / Klassen angepasst werden
- Streams können zur oder von der Konsole / Files / anderen Objekten bestehen

C++ Streams (2)

- Standard Input / Output:

`cin`	Standard Input-Stream
`cout`	Standard Output-Stream
`cerr`	Fehler-Output ohne Buffer
`clog`	Fehler-Output mit Buffer

  - Davon gibt es auch `wchar_t`-Versionen
  - `wcin, wcout, wcerr, wclog`
  - Definitionen im Header `<iostream>`
  - Für die Ausgabe muss << implementiert sein
  - Signatur:

```
ostream& operator<<(ostream &os, const UserType &u);
```

  - Sollte keine Member-Funktion sein
  - Wenn dann in `ostream`-Klasse

C++ Streams (3)

- Für die Eingabe muss >> implementiert sein
- Signatur

```
istream& operator>>(istream &is, UserType &u);
```

- Keine Member-Funktion
- Kein `const UserType &u`
- Alle Datenströme haben einen assoziierten Zustand
- Mit Member-Funktionen kann dieser abgefragt werden:

```
bool good()     // Next operation might succeed
bool eof()      // End of input seen
bool fail()     // Next operation will fail
                // (i.e. a previous operation failed)
bool bad()      // Stream is corrupted
iostate rdstate()                   // Get IO-state flags
void clear(iostatef = goodbit) // Set IO-state flags
void setstate(iostate f)            // Add f to flags
```

C++ Streams (4)

- Status-Flags sind in der Klasse `iobase` definiert

```
ios_base::badbit
ios_base::eofbit
ios_base::failbit
ios_base::goodbit
```

- Member-Funktion `rdstate()` kann benutzt werden, um die Flags zu lesen und weiter zu verarbeiten:

```
ios_base::iostates = cin.rdstate();
if (s & ios_base::badbit) {
  // Handle input errors.
}
```

- ... oder einfach die `bad()`, `fail()`, etc. Methoden verwenden

C++ Streams (5)

- Streams stellen Test-Operationen zur Verfügung:

```
void* operator();
bool operator!() const { return fail(); }
```

- Damit lassen sich elegant Schleifen bauen:

```
string word;
while (cin >> word) {
  // Do stuff with each word.
}
```

- >> gibt `istream&` zurück
- Dann Cast von `istream&` nach `void*`
- Der Rückgabewert hängt vom Status des Streams ab
- Gibt es nichts mehr zu lesen, wird `NULL` zurück gegeben

C++ Streams (6)

- Das Testen von Stream-Zuständen ist aufwändig
- Daher werfen Streams Exceptions
- Wann eine Exceptions geworfen werden soll, kann konfiguriert werden:

```
void exceptions(iostate except);
```

- Spezifikation der Zuständem die Exceptions werfen sollen
- Beispiel:

```
ios_base::badbit | ios_base::failbit
```

- Wenn der Stream nun in die angegebenen Zustände geht, wird die Exception `ios_base::failure` ausgelöst
- Um herauszufinden, welche Zustände Exceptions werfen gibt es auch eine Funktion:

```
iostate exceptions();
```

- Per Default sind alle Exceptions ausgeschaltet!!

C++ Streams (7)

- << und >> übernehmen die formatierte Ein- und Ausgabe
- Unformatierte Ein- und Ausgabe ist ebenfalls möglich
- `istreams` stellen die `get()`-Funktion zur Verfügung:

```
int get(); // Reads one character
istream& get(char &ch); // Reads a charater into ch
istream& get(char *p, int max)
istream& get(char *p, int max, char term)
istream& getline(char *p, int max)
istream& getline(char *p, int max, char term)
```

- `get()` und `getline()` lesen `max` Bytes oder bis zur Terminierung
- Terminierung in der Regel durch `newline`
- `get()` entfernt die Terminierung nicht
- Daher wird `getline()` bevorzugt verwendet

C++ Streams (8)

- Dateioperationen sind in C++ genau so einfach wie Ein- und Ausgabe zur Konsole
- Header:

```
#include <fstream>   // Read and write files
#include <ifstream>  // Read files
#include <ofstream>   // Write file
```

- Dateiname und Zugriffsmodi können im Konstruktor angegeben werden
- Beispiele:
  - Öffnen einer Textdatei zum Lesen

```
ifstream wordList("words.txt");
```

  - Öffnen einer Binärdatei, hänge neue Daten am Ende an

```
ofstream resData("result.dat", ios_base::binary,
    ios_base::append)
```

C++ Streams (9)

- `ios_base` definiert folgende Modi:

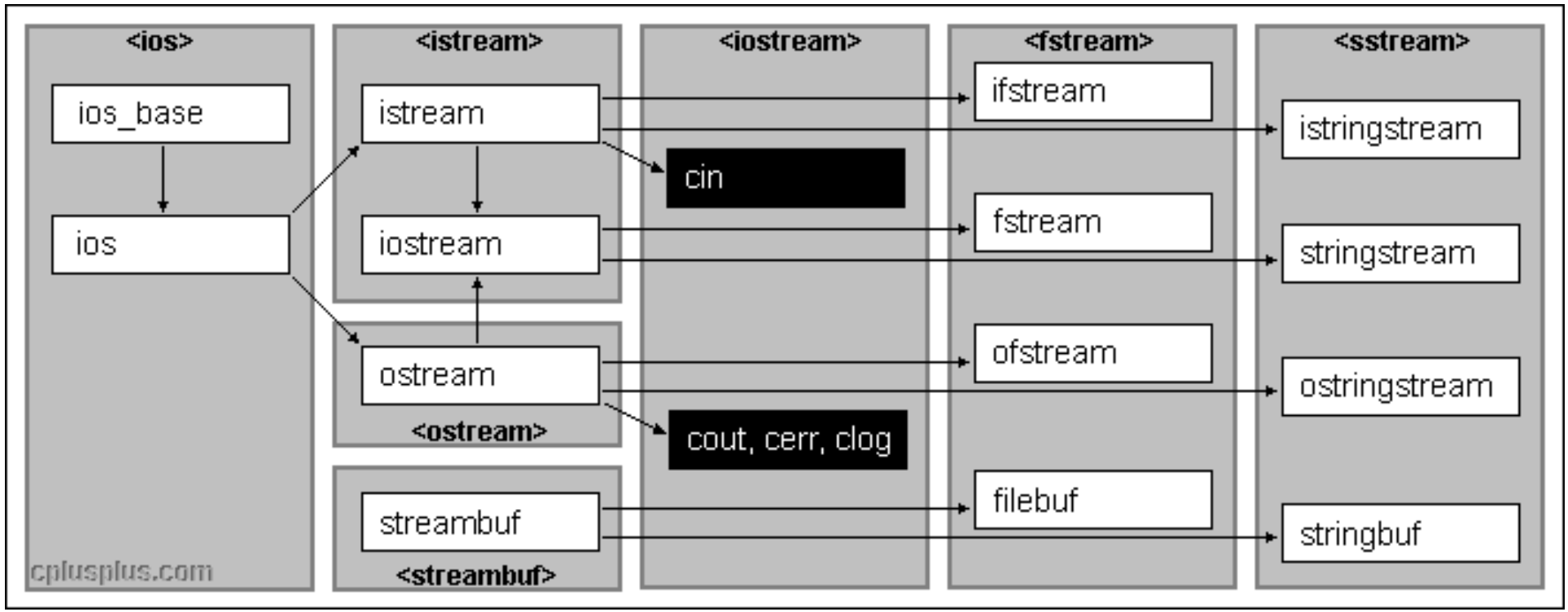
`app`	Öffnen und Anhängen
`ate`	Öffnen und zum Dateiende springen
`binary`	Binäre Ausgabe
`in`	Öffnen zum Lesen
`out`	Öffnen zum Schreiben
`trunc`	Dateigröße auf Null setzen

- Member-Funktionen zum Öffnen und Schließen:

```
void open(constchar *p, openmode m = out);
void close();
bool is_open();
```

C++ Streams (10)

- Klassendiagramm der C++ Streams



- Zusätzlich gibt es noch Streams für Strings

C++ Streams (11)

- `<sstream>` bindet Stringstreams ein
- Damit können Strings wie Streams gelesen und geschrieben werden
- Drei Typen (analog FileIO)
  - `stringstream` für Lesen und Schreiben
  - `istringstream` für Lesen
    - Damit lassen sich Parser einfach erstellen
  - `ostringstream` nur zum Schreiben
    - Nützlich für formatierte Ausgaben
    - Kann nicht überlaufen, d.h. wächst wenn benötigt
- Man kann auf die Daten eines Stringstreams zugreifen
  - Man kann dem `stringstream`-Konstruktor einen String mitgeben
  - Die `str()` Member-Funktion gibt den Inhalt des Streams als `string` zurück
  - `void str(const string&)` setzt den Inhalt des Stringstreams