Effective Analysis Programming - Part II

Hartmut Stadie, Christoph-Erdmann Pfeiler

GridKa school
September, 7th 2011

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Outline

Literature

Literature:

- Stroustrup: “The C++ Programming Language”, 3rd edition
- Sutter, Alexandrescu: “C++ Coding Standards”
- Press et al.: “Numerical Recipes 3rd edition”
- Meyers: “Effective C++” etc.
- ...

Read a book on programming!
You spend a lot of time writing code and should know how to do this!

Coding Guidelines

There are many ways to write C++ code. Use the right one!

Disclaimer

The following is heavily influenced by the book “C++ Coding Standards”.

Main idea:

Minimize the chance of bugs appearing in your code and find them quickly:

- use the compiler to find bugs
- write simple code, use clear designs
- always assume that the code will last long and be used by someone else

- Introduction
- Classes
  - Beispiel: Bruchrechnung
  - Introduction
  - Declaration and members
  - Constructors
  - Member functions
  - Access rights
  - Operators
  - In- und Output
  - Static members
- Rules for classes

Aufgabe: Bruchrechnung

Aufgabe:

Schreibe ein Programm zur Addition zweier Brüche!

Schema:

```
Bruch a und Bruch b einlesen
Bruch c = a + b berechnen
Bruch c ausgeben
```

Kodieren

Bruch besteht aus *int zaehler,nenner*

```
#include <iostream>
int main()
{
  //Bruch a und Bruch b einlesen
  int Azaehler, Anenner;
  std::cin >> Azaehler >> Anenner;
  int Bzaehler, Bnenner;
  std::cin >> Bzaehler >> Bnenner;
  //Bruch c = a + b berechnen
  int Czaehler, Cnenner;
  //?
  //Bruch c ausgeben
  std::cout << Czaehler << "/" <<  Cnenner << '\n';;
}
```

Kompilieren, testen, weiterdenken...

Kodieren II

Addition: $\frac{az}{an} + \frac{bz}{bn} = \frac{az \cdot bn + bz \cdot an}{an \cdot bn}$

```
  //Bruch a und Bruch b einlesen
  int Azaehler, Anenner;
  std::cin >> Azaehler >> Anenner;
  int Bzaehler, Bnenner;
  std::cin >> Bzaehler >> Bnenner;
  //Bruch c = a + b berechnen
  int Czaehler = Azaehler * Bnenner +
                        Bzaehler * Anenner;
  int Cnenner = Anenner * Bnenner;
}
```

Kompilieren, testen, weiterdenken...

Kodieren III

Ergebnis kürzen:
Zähler und Nenner durch ihren größten gemeinsamen Teiler dividieren.

```
int ggT(int a, int b) {
  if(b == 0) return a;
  return ggT(b, a % b);
}
int main()
{
   ...
  int Czaehler = Azaehler * Bnenner +
                       Bzaehler * Anenner;
  int Cnenner = Anenner * Bnenner;
  int d = ggT(Czaehler ,Cnenner);
  Czaehler /= d;
  Cnenner /=d;
}
```

Kompilieren, testen, weiterdenken...fertig

Klassen

Klassen:

Benutzer definierter Typ
Typ mit Membern(interne Variablen und Funktionen)
verschiedene Zugriffsrechte:

- **public**: alle dürfen Zugreifen
- **private**: nur innerhalb der Klasse benutzbar

struct : Member public als Default
class : Member private als Default
Objekt: Instanz von Typ **class T**
this: Zeiger auf Objekt, für das die Funktionen aufgerufen wurde

Deklaration und Member

Beispiel Bruch:

```
class Bruch
{
 public:
  int z,n;//Datenmember
};
```

Addier-Funktion

```
Bruch addiere(Bruch a, Bruch b)
{
  Bruch c;
  c.z = a.z * b.n + b.z * a.n;
  c.n = a.n * b.n;
  int d = ggT(c.z ,c.n);
  c.z /= d;
  c.n /= d;
  return c;
}
```

Hauptprogramm

```
int main()
{
  //Bruch a und Bruch b einlesen
  Bruch a;
  std::cin >> a.z >> a.n;
  Bruch b;
  std::cin >> b.z >> b.n;
  Bruch c;
  std::cin >> c.z >> c.n;
  //Bruch c = a + b +c berechnen
  Bruch d = addiere(a,b);
  Bruch e = addiere(d,c);
  std::cout << e.z << "/" << e.n << '\n';
}
```

weitere Wünsche

weitere Wünsche:

- einfache Initialisation
- einfache Ausgabe
- automatisches Kürzen
- ...

Konstruktoren und Destruktoren

Konstruktor:

- wird bei der Erzeugung von Variablen aufgerufen
- definiert, wie ein Objekt initialisiert wird
- erlauben Typumwandlungen

Destruktor:

wird bei der Zerstörung einer Variablen aufgerufen

Konstruktoren

```
class Bruch{
public:
  Bruch();//default constructor
  Bruch(int n, int z);//specific constructor
  Bruch(const Bruch& b);//copy constructor
  ~Bruch();//destructor
};
...
Bruch b;
Bruch a(3,2);
```

Bruch mit Konstruktoren

```
class Bruch
{
public:
  Bruch();
  Bruch(int nz, int nn);

  int z,n;
};
Bruch::Bruch()
{
  z = 0;
  n = 1;
}
Bruch::Bruch(int nz, int nn)
{
  z = nz;
  n = nn;
}
Bruch addiere(Bruch a, Bruch b)
{
  Bruch c(a.z * b.n + b.z * a.n,a.n * b.n);
  int d = ggT(c.z ,c.n);
  c.z /= d;
  c.n /= d;
  return c;
}
```

Kopieren von Objekten

ohne *copyconstructor*: kopiere alle Member 1 zu 1;
oder eigener Konstruktor:

```
class Bruch
{
public:
  Bruch();
  Bruch(int nz, int nn);
  Bruch(const Bruch& b);

  int z,n;
};

Bruch::Bruch(const Bruch& b)
{
  z = b.z;
  n = b.n;
  //std::cout << "copy called fuer " << z << "/" << n << "\n";
}
```

Konstruktoren und Typumwandlung

Konstrukoren definieren auch die Typumwandlung(*casting*)
Beispiel: int $\rightarrow$ Bruch

```
class Bruch {
 Bruch(int nz);
 ...
};
Bruch::Bruch(int nz)
{
  z = nz;
  n = 1;
}
...
int main() {
  Bruch f = 7;
}
```

Wann eigene Konstruktoren und Destruktoren?

Beispiel: Objekt mit eigenem Speicher

```
class Vector
{
private:
  int dim;
  double* val;
public:
  Vector(int dimension);
  void set(int i, double d);
};
Vector::Vector(int dimension) : dim(dimension)
{
  val = new double[dim];
  for(int i = 0; i < dim ; ++i) val[i] = 0;
}
void Vector::set(int i, double d)
{
  val[i] = d;
}
int main()
{
  Vector v(6);
  v.set(4,3.13);
}
```

Eigener Destruktor fehlt!

Speicher muss wieder freigegeben werden!

```
class Vector
{
private:
  int dim;
  double* val;
public:
  Vector(int dimension);
  ~Vector();
  void set(int i, double d);
};
Vector::Vector(int dimension)
 : dim(dimension)
{
  val = new double[dim];
  for(int i = 0; i < dim ; ++i) val[i] = 0;
}
Vector::~Vector()
{
  delete [] val;
}
```

Eigener Copyconstructor fehlt!

Einfache Kopie aller Member, dupliziert nicht das Feld!

```
class Vector
{
private:
  int dim;
  double* val;
public:
  Vector(int dimension);
  ~Vector();
  Vector(const Vector& v);
  void set(int i, double d);
};
Vector::Vector(const Vector& v)
{
  dim = v.dim;
  val = new double[dim];
  for(int i = 0; i < dim ; ++i) v.val[i] = 0;
}
```

Unbeabsichtigte Typumwandlung

Konstruktor erlaubt: int $\rightarrow$ Vector
explicit verbietet die Benutzung eines Konstruktors zur Typumwandlung.

```
class Vector
{
private:
  int dim;
  double* val;
public:
  explicit Vector(int dimension);
  ~Vector();
  Vector(const Vector& v);
  void set(int i, double d);
};
```

Memberfunktionen

Memberfunktionen werden für ein bestimmtes Objekt aufgerufen.
Beispiel:

```
class Bruch
{
public:
  ...
  void kuerze();
};
void Bruch::kuerze() {
  int d = ggT(z ,n);
  z /= d;
  n /= d;
}
Bruch addiere(Bruch a, Bruch b)
{
  Bruch c(a.z * b.n + b.z * a.n,a.n * b.n);
  c.kuerze();
  return c;
}
```

Memberfunktionen II

addiere:

```
class Bruch
{
public:
  Bruch addiere(Bruch b);
};

Bruch Bruch::addiere(Bruch b)
{
  Bruch c(z * b.n + b.z * n,n * b.n);
  c.kuerze();
  return c;
}

int main()
{...
  Bruch d = a.addiere(b);
}
```

Zugriffsrechte

Damit der Bruch immer richtig gekürzt ist, sollte niemand direkt den Zähler oder Nenner ändern dürfen.
Also: Setze *z,n private*

```
class Bruch
{
public:
  Bruch();
  Bruch(int nz, int nn);
  Bruch(const Bruch& b);
  Bruch addiere(Bruch b);
  void print();
private:
  void kuerze();
  int z,n;
};
```

Operatoren

Idee: Bruch c = a + b;

```
in Klasse:
Bruch operator+(Bruch b);
...
Bruch Bruch::operator+(Bruch b) {
  return Bruch(m_z * b.m_n + m_n * b.m_z,
               m_n * b.m_n);
}
...
Bruch erg = b1 + b2;
Bruch erg = b1.operator+(b2);
```

Ausgabe

std::cout

- ist vom Typ *std::ostream*
- benutzt:

```
std::ostream& operator<<(std::ostream& os,
                         Bruch b);
```

```
//mit Selbstreferenz
std::ostream& operator<<(std::ostream& os, Bruch b) {
  return os << b.z << '/' << b.n;
}

std::cout << "Bruch:" << erg2 << '\n';
```

Muss *friend* von Bruch sein!

Einlesen

std::cin

- ist vom Typ *std::istream*
- benutzt:

```
std::istream& operator>>(std::istream& is,
                         Bruch& b);
```

```
std::istream& operator>>(std::istream& is,Bruch& b) {
  is >> z >> n;
  return is;
}
int main() {
Bruch a;
std::cin >> a;
}
```

Muss *friend* von Bruch sein!

Statische Member

statischer Member

Member, der nur einmal für alle Objekte des neuen Typs existieren soll.

```
class Bruch{
  static int ggT(int a, int b);
 };
 int g = Bruch::ggT(5,3);
```

Design mit Klassen

Wann Klassen und wann Funktionen?

- Beschreibe Aufgabe
  - Substantive werden zu Objekten (Klassen)
  - Verben werden zu Funktionen der entsprechenden Klassen
- Beschreibe Objekte
  - ist beschreibt Typ
  - hat beschreibt Member

Benutzer interessieren nur die Memberfunktionen!!!

- Introduction
- Classes
- Rules for classes
  - Prefer minimal classes to monolithic classes
  - Prefer composition to inheritance
  - Prefer providing abstract interfaces
  - Avoid providing implicit conversions
  - Make data member private, except in behaviorless aggregates
  - Do not give away your internals
  - Prefer initialization to assignment in constructor
  - Avoid calling virtual functions in constructor or destructor
  - Copy and destroy consistently
  - Explicitly enable or disable copying
  - Avoid slicing consider **clone**

Prefer minimal classes to monolithic classes

“Divide and conquer”:

“Small classes are easier to write, get right, test, and use. They are also more likely to be usable in a variety of situations. Prefer such small classes that embody simple concepts instead of kitchen-sink classes that try to implement many and/or complex concepts.”

Prefer composition to inheritance

“Avoid inheritance taxes”:

“Inheritance is the second-tightest coupling relationship in C++, second only to friend. Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.”

Prefer providing abstract interfaces

“Love abstract art”:

“Abstract interfaces help you focus on getting an abstraction right without mudding it with implementation or state management details. Prefer to design hierarchies that implement abstract interfaces that model abstract concepts.

Avoid providing implicit conversions

Summary:

"Not all change is progress: Implicit conversions can often do more damage than good."

check Map m = 1;

Make data members private, except in behaviorless aggregates

Summary:

"They are none of your caller's business: Keep data members **private**. Only in the case of simple C-style **struct** types that aggregate a bunch of values but do not pretend to encapsulate or provide behavior, make all data members **public**.Avoid mixes of public and nonpublic data."

Don’t give away your internals

Summary:

“Do not volunteer too much: Avoid returning handles to internal data managed by your class, so clients will not uncontrollably modify state that your object thinks it owns.”

Prefer initialization to assignment in constructor

Summary:

"Set once, use everywhere: In constructors, using initialization instead of assignment to set member variables prevents needless run-time work and takes the same amount of time."

Avoid calling virtual functions in constructor or destructor

Summary:

"Virtual function only *virtually* always behave virtually."

Copy and destroy consistently

Summary:

"What you create, also clean up: If you define any of the copy constructor, copy assignment operator, or destructor, you might need to define one or both of the others."

Explicitly enable or disable copying

Summary:

"Copy consciously: Knowlingly choose among using the compiler-generated copy constructor an assignment operator, writing your own versions, or explicitly disabling both if copying should not be allowed."

Avoid slicing, consider **clone**

Summary:

"Object slicing is automatic, invisible, and likely to bring wonderful polymorphic designs to a screeching halt. In base classes, consider disabling the copy constructor and copy assignment operator, and instead supplying a virtual **Clone** member function if client need to make polymorphic copies."