ILV Datenstrukturen und Algorithmen

04: Function/operator overloading, static members

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

Licensed under CreativeCommons Attribution-NonCommercial-ShareAlike 3.0

Kapitel

Function and operator overloading

Function overloading

```
1  void print(char i);
2  void print(double d);
3  void print(int array[], unsigned size);
4
5  void test()
6  {
7    int arr[] = {1, 2, 3};
8    print('a');    //calls print(char)
9    print(2.0);    //calls print(double)
10   print(arr, 3);  //calls print(int[], unsigned)
11 }
```

- C++ erlaubt Funktionen mit gleichem Namen mehrfach zu deklarieren.
- Der Compiler entscheidet anhand der Argumenttypen, welche Funktion aufgerufen werden soll.
  - Der Rückgabetyp ist nicht relevant.
  - Man könnte nicht noch eine Funktion `int print(char i);` deklarieren.
- Demo...

Function overloading

- Beim Aufruf einer Funktion muss es also eine Funktion mit passender Signatur geben.
  - Signatur = Funktionsname + Folge Parametertypen.
  - Folgende Funktionen haben die *gleiche* Signatur:
    - `void test(int param1, char param2);`
    - `int test(int p1, char p2);`
- Man spricht vom „Überladen“ von Funktionen.
  - Überschreiben und Überladen sind zwei verschiedene Dinge! Beim Überschreiben *muss* die Signatur gleich bleiben.
  - Bisher schon bekannt: Überladen von Konstruktoren.

Default arguments

```
1  /* Convert unsigned to string. Base=10 is decimal system.*/
2  string uintToString(unsigned number, unsigned base=10);
3
4  // in cpp file:
5  string uintToString(unsigned number, unsigned base)
6  {
7    ...
8  }
9
10 void test()
11 {
12   cout << uintToString(23432) << endl;
13 }
```

- C++ erlaubt die Angabe von Standardwerten für Argumente.
- Parameter, für die default arguments definiert sind, müssen nicht angegeben werden.
- Default arguments sind nur für die letzteren Argumente erlaubt.
  - `void f(int a, int b=0, int c, int d=0);` ist nicht erlaubt. Warum?
- Die default arguments werden nur in der Deklaration angegeben.

Operator overloading

- C++ definiert für viele Datentypen zugehörigen Operatoren auf natürliche Weise, etwa `a+b` für zwei Integer `a, b`.
- Für eigene Datentypen sind Operatoren selbstverständlich nicht definiert.
- C++ bietet aber die Möglichkeit Operatoren zu „überladen“.

```
1 Vector u, v, w;
2 double a;
3 w = u + v;
4 w += a*v;
```

Code demo...

Operator overloading

- C++ bietet verschiedenste Operatoren zum Überladen an:
  - unäre Operatoren: `~ - [] ! & ++ --` usw.
  - binäre Operatoren: `+ - * / << >> < > <= >= != == && ||` usw.
  - Zuweisungen: `= += -= *= <<=` usw.
  - function call Operator: `()`
  - weitere Operatoren: `, -> ->* new` usw.
- Name der Operatorfunktion: `operator` und dem Operator-Zeichen.
  - `Vector operator+(const Vector& a, const Vector& b);`
  - `bool operator==(const Color& a, const Color& b);`

Operator overloading

- Kann auch innerhalb einer Klasse überladen werden.

```
1  class Vector
2  {
3    public:
4      Vector operator+(const Vector& second);
5  };
6  void test()
7  {
8    Vector a, b, c;
9    c = a+b;       // this is the same as...
10   c = a.operator+(b);
11 }
```

Assignment operator: Ergänzungen

- Wissen schon: Wenn Klassen Speicher reservieren, dann muss `operator=` überladen werden.
  - Sonst: double free im Destruktor.
  - Parametertyp ist eine konstante Referenz!
- `operator=` kann `void` zurückliefern, muss aber nicht.
  - `a = b = c;` geht nicht, wenn `operator= void` liefert.
- In der Praxis wichtig: Achte auf self assignment!

```
1  IntArray& IntArray::operator=(const IntArray& copy)
2  {
3    // Check this, or we free copy.mem below!
4    if( this == &copy)
5      retun *this;
6
7    delete[] mem;
8    size = copy.size;
9    mem = new int[size];
10   for(unsigned i=0; i<size; ++i)
11     mem[i] = copy.mem[i];
12 }
```

Operator overloading: Ratschläge

- Java und viele andere Sprachen verbieten operator overloading, weil Programmierer dazu tendieren, alle möglichen Operatoren zu überladen: hinter jedem Operator-Aufruf kann dann alles Mögliche stecken.
- Daher: verwende operator overloading nicht exzessiv, sondern nur, wenn die Bedeutung unmittelbar klar ist!
  - `Vector::operator<=(const Vector& v)` ist schlecht.
  - `IntArray::operator+=(int value)` ist ok.
- Wenn `operator==` überladen wird, dann soll auch `operator!=` überladen werden.
- Wenn `operator<` überladen wird, dann sollen auch `operator==`, `operator!=`, `operator<=`, `operator>=`, `operator>` überladen werden.

f(0)

Was macht `f(0)`?

- Ist ein `operator()` für das Objekt `f` definiert?
- Gibt es einen Funktion-Pointer `f`?
- Gibt es Basisklassen, welche eine Funktion `f` definieren?
- Gibt es eine Funktion `f` mit default Argumenten derart, dass man nur ein Argument angeben muss?
- Gibt es eine Klasse `f`, die einen Konstruktor hat, welcher ein `int` nimmt?

Kapitel

Static member functions

Static member functions: Motivation

```
1 class Color
2 {
3     /** Red, Green, Blue */
4     double r, g, b;
5   public:
6     Color(double r, double g, double b);
7 };
```

Wir möchten `Color` instantiieren durch Angabe von

1. HSV-Werte (Hue, Saturation, Value),
2. Greyscale-Wert, etc.

Für die Variante 1 können wir keinen eigenen Konstruktor erstellen, da schon ein Konstruktor mit der Signatur `Color(double, double, double)` existiert.

Static member functions: Motivation

```
1  class Color
2  {
3      double r, g, b;
4    public:
5      Color(double r, double g, double b);
6      static Color fromHSV(double h, double s, double v);
7  };
8
9  //cpp file:
10 Color Color::fromHSV(double h, double s, double v)
11 {
12   // ...
13 }
```

Lösung: static member functions.

- Gewöhnliche Funktionen gehören zu einem Objekt: `double a = rect.area();`
- Statische Funktionen gehören zu einer Klasse.
- Statische Funktionen werden ohne Objekt aufgerufen, sondern durch Angabe der Klasse.

```
Color red = Color::fromHSB(1.0, 1.0, 1.0);
```

Off-topic: HSV to RGB

Seien $H \in [0, 360), S \in [0, 1], V \in [0, 1]$ gegeben.

$$C := V \cdot S$$
$$M := V - C$$
$$X := C \cdot \left(1 - \left|\frac{H}{60} \mod 2 - 1\right|\right)$$
$$(R, G, B) = (M, M, M) + \begin{cases} (C, X, 0) & \text{falls } H \in [0, 60) \\ (X, C, 0) & \text{falls } H \in [60, 120) \\ (0, C, X) & \text{falls } H \in [120, 180) \\ (0, X, C) & \text{falls } H \in [180, 240) \\ (X, 0, C) & \text{falls } H \in [240, 300) \\ (C, 0, X) & \text{falls } H \in [300, 360) \end{cases}$$

Achtung: $\frac{H}{60}$ muss nicht ganzzahlig sein; z.B. 5.46 mod 2 würde 1.46 ergeben.

Static members: facts

- Man kann auch Attribute als `static` deklarieren.
  - Eine static member Variable hat genau einen Platz im Speicher. Bei non-static member Variablen wird pro Objekt Speicher aufgewendet.
  - Beispiel: Definieren von Standard-Farben als statische Attribute:

```
1 class Color
2 {
3   static const Color red;
4 };
5 const Color Color::red = Color(1,0,0);   // cpp file
```

- Static members befinden sich auf „Klassenebene“, nicht auf „Objektebene“:
  - Es gibt keinen `this` Pointer in statischen Funktionen.
  - Statische Funktionen können nicht auf nicht-statische Funktionen oder nicht-statische Attribute zugreifen.
  - Statische Funktionen können weder `virtual` noch `const` sein. Statische Attribute können allerdings `const` definiert werden.
  - Statische und nicht-statische Funktionen können sich nicht gegenseitig überschreiben.

Fazit: static member functions verhalten sich in etwa wie in Klassen gekapselte globale Funktionen mit Zugriffsschutz.

Anwendung: singleton design pattern

- Gamma et al: *Ensure a class only has one instance, and provide a global point of access to it.*
  - Es soll nur eine Instanz einer Klasse geben.
  - Die Klasse soll aber durch Vererbung/Polymorphismus erweiterbar bleiben.
- Beispiel:
  - Ein Klasse Logger zum Loggen von Nachrichten durch das Programm.
  - Eine Klasse Configuration, welche sich Applikations-Einstellungen merkt.
- Besser als eine globale Instanz der Klasse oder eine Klasse mit ausschließlich statischen Members:
  - Verschmutzt den globlen Namensraum nicht.
  - Bleibt erweiterbar.
  - Lazy allocation.
  - C++ definiert nicht, in welcher Reihenfolge globale Variablen oder statische Members initialisiert werden! Wird für die Initialisierung einer globalen Variable Funktionalität einer anderen globalen Variable benötigt, führt das schnell zum Absturz.
- Implementierung:
  - Speichere die eine Singleton-Instanz als statische member Variable.
  - Wie verhindert man weitere Instanzen? Wie erzeugt man die eine Instanz per lazy allocation?
    - Siehe neues Übungsblatt.