Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

C++ Schreibweisen (1)

- Code mit vielen Variablen kann leicht unübersichtlich werden
- „Ungarische Notation“ kodiert den Typ im Variablennamen

bool / unsigned char	b	vorzeichenloser 8-Bit Wert, bool-Wert
char	c oder ch	vorzeichenbehafteter 8-Bit Wert, ASCII-Zeichen
unsigned short	w	vorzeichenloser 16-Bit Wert
short / integer	n	vorzeichenbehafteter 16-Bit Wert, Integer
unsigned long	dw	vorzeichenloser 32-Bit Wert
long	l	vorzeichenbehafteter 32-Bit Wert
float	f	32-Bit Gleitkommazahl
double	d	64-Bit Gleitkommazahl
double float	df	64-Bit Gleitkommazahl (VC++), 96 Bit bei MinGW

C++-Schreibweisen (2)

class	C	Klasse oder Instanz einer Klasse
union	u	Variante (union) oder Instanz einer Variante
struct	str	Definition oder Instanz einer Struktur
enum	e	Enumeration oder Instanz einer Enumeration

Zusatz-Präfixe, stehen immer in Kombination mit einem anderen Präfix und an erster Stelle:

sz	Zeichenkette mit ‘0‘ abgeschlossen
p	Zeiger (Pointer)
a	Feld (Array)
m_	Member (einer Klasse)

```
class CPoint {
  double m_dxCoord, m_dyCoord;   // Data-members
...
```

C++ Schreibweisen (3)

- Beispiel:

```
// Sort the item in reverse alphabetical order.
static int CALLBACK
MyCompareProc(LPARAM lParam1, LPARAM lParam2, LPARAM
lParamSort)
{
   CListCtrl* pListCtrl = (CListCtrl*) lParamSort;
   CString    strItem1 = pListCtrl->GetItemText(lParam1, 0);
   CString    strItem2 = pListCtrl->GetItemText(lParam2, 0);
   TRACE("\nSTRING 1: %s\nSTRING 2: %s",strItem1,strItem2);
   return strcmp(strItem2, strItem1);
}
```

Hungarian Notation

Insist on carrying outright orthogonal information in your Hungarian warts. Consider this real world example:

```
a_crszkvc30LastNameCol
```

It took a team of maintenance engineers nearly 3 days to figure out that this whopper variable name described a const, reference, function argument that was holding information from a database column of type Varchar[30] named "LastName" which was part of the table's primary key. When properly combined with the principle that "all variables should be public" this technique has the power to render thousands of lines of source code obsolete instantly!

Hungarian Notation

Use to your advantage the principle that the human brain can only hold 7 pieces of information concurrently. For example code written to the above standard has the following properties:

- a single assignment statement carries 14 pieces of type and name information.
- a single function call that passes three parameters and assigns a result carries 29 pieces of type and name information.
- Seek to improve this excellent, but far too concise, standard. Impress management and coworkers by recommending a 5 letter day of the week prefix to help isolate code written on 'Monam' and 'FriPM'.
- It is easy to overwhelm the short term memory with even a moderately complex nesting structure, especially when the maintenance programmer can't see the start and end of each block on screen simultaneously.

Design-Patterns

- Manche Probleme treten in der Softwareentwicklung immer wieder auf
- Für einige Probleme lassen sich gewisse „Lösungs-Rezepte“ vorgeben, die man immer wieder anwenden kann
- Entwurfsmuster bzw. „design patterns“
- Anforderungen:
  - Es sollten mehrere Probleme auf einmal gelöst werden
  - Das Konzept sollte erprobt und verstanden sein
  - Leicht umzusetzen sein
  - Paxisnah und erprobt sein
- Hier nur ein paar wichtige Konzepte in aller Kürze
- Ansonsten: Veranstaltungen zum Software-Engineering besuchen!

Singleton (1)

- Oftmals ist es sinnvoll nur ein Objekt zu haben, das eine bestimmte Aufgabe erfüllt.
- Beispiele:
  - Schreiben von Daten in Dateien
  - Verwaltung globaler Datenobjekte über mehrere Klassen
- Anforderung: Es muss sicher gestellt sein, dass es nur eine einzige Instanz eines solchen Objektes im Programm gibt!

➡Wie realisiert man so was in C++?

➡Man macht den Konstruktor einer Klasse private oder protected!

Singleton (2)

- (Minimal)Beispiel:

```
class Singleton
{
  Singleton();
  static Singleton* m_instance;

public:
  static Singleton* getInstance();
};
```

```
Singleton::m_instance = 0;

Singleton* Singleton::getInstance()
{
  if(Singleton::m_instance == 0)
  {
Singleton::m_instance = new
Singleton;
      return Singleton::m_instance;
  }
  else
  {
      return Singleton::m_instance;
  }
}
```

Fabrik(methode)

- Liefert in Abhängigkeit eines gegebnen Parameters spezialisierte Objekte.
- Beispiel:

```
Block* BlockFactory::createNewBlock(int blockType)
{
   Block* block = 0;
   switch(blockType)
   {
   case 0 : block = new SquareBlock;      break;
   case 1 : block = new LeftLBlock;       break;
   case 2 : block = new LZetBlock;        break;
   case 3 : block = new RightLBlock;      break;
   case 4 : block = new RZetBlock;        break;
         default: break;
   }
   return block;
}
```

- Oftmals als Singletons realisiert!

Model-View-Controller (MVC)

- Trennung von Datenmodell, Darstellung und Operationen auf den Daten
- Das Modell enthält die aktuellen Daten und evtl. erlaubt Operationen darauf
- Die „View“ stellt den aktuellen Zustand der Daten dar, erlaubt evtl. Benutzerinteraktionen
- Macht aber keine direkte Datenverarbeitung
- Der Kontroller nimmt Eingaben einer oder mehrere Views da und modifiziert die Daten
- View ist also direkt auf die Darstellung ausgelegt
- Datenlogik wird auf den Controller ausgelagert
- Häufig über Visitors oder Signale realisiert
- Wichtiges Konzept bei der Programmierung von grafischen Oberflächen

Gliederung

C++ Konstanten (1)

- Von der Verwendung von `#define` für Konstanten in C++-Programmen wird dringend abgeraten
- `const` Schlüsselwort zur Definition von Konstanten

```
const int MaxWidgets = 1000;
```

- Zu Verwenden, wie eine Variable
- Sagt dem Compiler, dass diese Variable sich nicht ändern wird
- Enthält nun eine Typ-Information
- Compiler sorgt während des Übersetzens dafür, dass die Variable konstant bleibt
- Fehlermeldungen des Compilers
- Neu: ANSI C (1999) hat `const` Konzept übernommen (vgl. Folie 132)

C++ Konstanten (2)

- `const` ist sehr nützlich, wenn mit Referenzen gearbeitet wird
  - Erlaubt nicht, dass sich am Referent etwas ändert
  - Unerwünschte Seiteneffekte werden vermieden
- Schnell und gefährlich:

```
double Point::distanceTo(Point &p)
{...}
```

- Schnell und sicher:

```
double Point::distanceTo(const Point &p)
{...}
```

- Sollte immer verwendet werden
- Copy-Konstruktor sollte immer ein `const` Objekt übergeben bekommen:

```
Point(const Point &p);
// Don't change the original!
```

C++ Konstanten (3)

- `const` muss sowohl in der Methodendeklaration (Header-File) als auch in der Methodenimplementierung (`.cc`-File) erscheinen
- Beispiel:

```
double Point::distanceTo(const Point &p) {
  double dx = x_coord - p.getX();
  double dy = y_coord - p.getY();
  return sqrt(dx * dx+ dy * dy);
}
```

- Wie weiß der Compiler, dass `getX()` und `getY()` nicht den Punkt verändern werden?
- Der Compiler wird sich beschweren, wenn wir dies nicht tun
- Wir müssen dem Compiler sagen, dass `getX()` und `getY()` nichts verändern werden

C++ Konstanten (4)

- Wir spezifizieren `const` nach der Methodendeklaration, wenn nichts am internen Zustand der Klasse geändert wird

```
class Point {
  ...
  double getX() const; // These don't change
  double getY() const; // the Point's state.
  ...
};
```

- Dieses `const` muss wiederum in Header- und .cc-Datei angegeben werden
- `const` kann aber noch an einer weiteren Stelle stehen ☺

```
const Point & getMyPointRef();
```

C++ Konstanten (5)

- `const Point & getMyPointRef();`
- ... bedeutet, dass der Rückgabewert sich nicht ändern darf
- ... bedeutet hier, dass der Referent nicht modifiziert werden darf
- Es geht auch ohne Referenz:

```
const Point getMyPointRef();
```

- Beispiel:

```
class X {
  int i;            // private
public:
  X(int _i = 0); // Constructor
  void modify(); // Mutator
}
X::X(int _i) { i = _i; }
void X::modify() { i++; }
```

C++ Konstanten (6)

```
class X {
  int i;
public:
  X(int _i = 0);
  void modify();
};

X::X(int _i)
{ i = _i; }
void X::modify() { i++; }


// main program
int main() {
  f1() = X(1);  // O.K. non-const ref
  f1().modify();// O.K.

  f3(f2());     // Error or Warning
  f2() = X(1);  // Error or Warning
  f2().modify();// Error or Warning
}
```

```
// Global functions
X f1() {
  return X();
}

const X f2() {
  return X();
}

void f3(X &x) {
 x.modify();
}
```

Gliederung

C++ Dynamische Speicherbereitstellung (1)

- In C gab es `malloc()` / `calloc()` um Speicher bereitzustellen und `free()` um ihn wieder frei zu geben
- Die C++-Befehle dazu sind `new` und `delete`
- `new` und `delete` sind keine Funktionen
- Rückgabewert von `new` sind Pointer mit einem Typ
- Beispiel:

p ist ein Pointer zu der Instanz von Point

```
Point *p = new Point(3.5, 2.1);
p->setX(3.8);                  // Use the point...
delete p;                      // Free the point
```

- Lokale Variablen: Destruktor wird automatisch aufgerufen

```
void doStuff() {
  Point a(-4.75, 2.3); // Make a Point...
}
```

Destruktor wird hier aufgerufen

C++ Dynamische Speicherbereitstellung (2)

- Allokierte Objekte müssen selbst aufgeräumt werden
- Beispiel:

```
void leakMemory() {
   Point *p = new Point(-4.75, 2.3);
   ...
}
```

Der Pointer ist weg, das Objekt ist noch vorhanden!

- Das Objekt wird erst durch einen expliziten Aufruf von `delete ptrToObj` gelöscht
- Arrays von Objekten (statisch):

```
Point tenPoints[10];          // Index 0..9
tenPoints[3].setX(21.78);     // Fourth element
```

- Default Destruktor wird für jedes Objekt aufgerufen

C++ Dynamische Speicherbereitstellung (3)

- Arrays von Objekten (dynamisch mit `new`-Operator):

```
Point *somePoints = new Point[8];   // Index 0..7
somePoints[5].setY(-14.4);          // 6th element
```

- Dynamisch erzeugte Arrays müssen mit dem `delete[]`-Operator gelöscht werden:

```
delete[] somePoints; // Clean up my Points
```

- Compiler verhindert ein einfaches `delete` nicht!
- Man kann `new/delete[]` auch auf einfache Datentypen anwenden:

```
int numValues = 35;
int *valArray = new int[numValues];
```

- Felder sind nicht initialisiert

C++ Dynamische Speicherbereitstellung (4)

- Wenn eine Klasse dynamisch allokierten Speicher verwendet:
  - Falls möglich: Speicherbereitstellung im Konstruktor
  - Speicherfreigabe im Destuktor
- Beispiel:

```
// Initialize a vector of n floats.
FloatVector::FloatVector(int n)
{
  numElems = n;
  elems = new float[numElems];
}

// Clean up the float-vector.
FloatVector::~FloatVector()
{
  delete[] elems; // Release the memory for the array.
}
```

C++ Dynamische Speicherbereitstellung (5)

- Nun räumt `FloatVector` selbst auf:

```
float calculate() {
  FloatVector fvect(100); // Use our vector
  ...
}
```

- `fvect` ist eine lokale Variable
- Der Destruktor gibt dynamisch angeforderten Speicher wieder frei
- Hier wollen wir bestimmt nicht den default-Destruktor!

C++ Copy-Konstruktor (1)

- Beispiel Punkt-Klasse (`Point.hh`)

```
// A 2D point class!
class Point {
  double x_coord, y_coord;   // Data-members
public:
  Point();                   // Constructors
  Point(double x, double y);
  ~Point();                  // Destructor
  double getX();             // Accessors
  double getY();
  void setX(double x);       // Mutators
  void setY(double y);
};
```

- Nun möchten wir eine Kopie eines Punktes erstellen

```
Point p1(3.5, 2.1);
Point p2(p1);                // Copy p1.
```

C++ Copy-Konstruktor (2)

- Dies funktioniert wegen des Copy-Konstruktors
- Der Copy-Contruktor wird immer dann aufgerufen, wenn eine Kopie eines Objektes erzeugt wird
- Signatur:

```
MyClass(MyClass &other);   // Copy-constructor
```

- Argument ist eine Referenz
- Aufrufe sehen z.B. so aus:

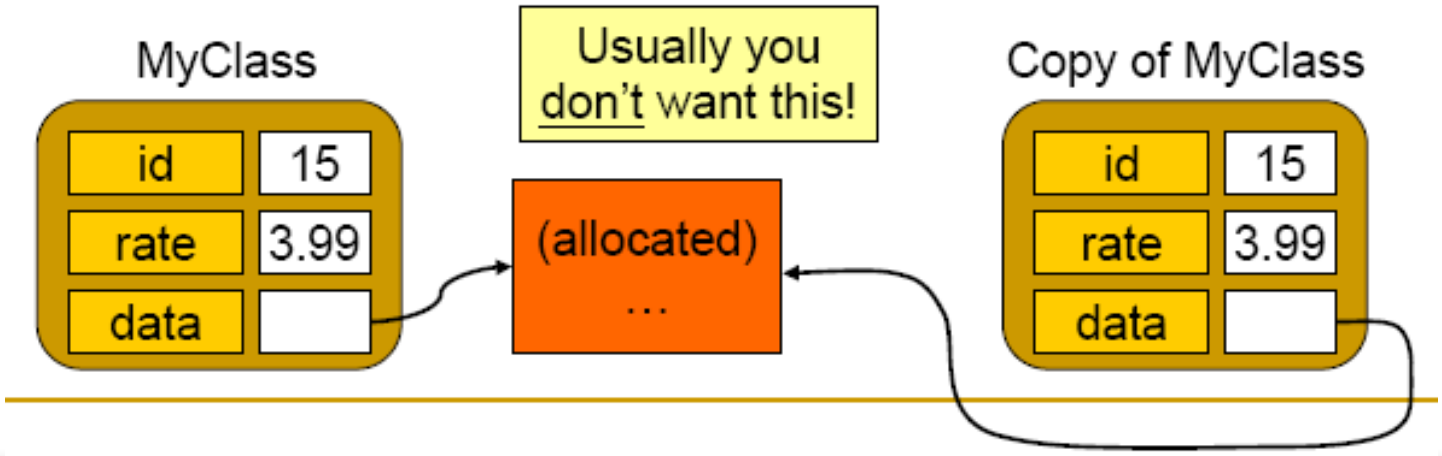
```
MyClass(MyClass other);
```

- In der Klasse Point gibt es keinen Copy-Konstruktor
- Der Compiler fügt automatischen einen hinzu
- „Default Copy-Konstruktor“

C++ Copy-Konstruktor (3)

- Der Compiler stellt einen Default Copy-Konstruktor zur Verfügung
- Dieser kopiert einfach alle einfachen Datentypen im Objekt
- Es handelt sich um eine „flache Kopie

  - Wenn im Objekt ein Pointer auf ein Array/Objekt steht, wird nur der Pointer kopiert, nicht das Array/Objekt
  - Das Originalobjekt und die Kopie teilen sich jetzt Speicher!
  - Wenn der Destruktor dynamisch zugeteilten Speicher löscht, führt das zu einem Programmabsturz



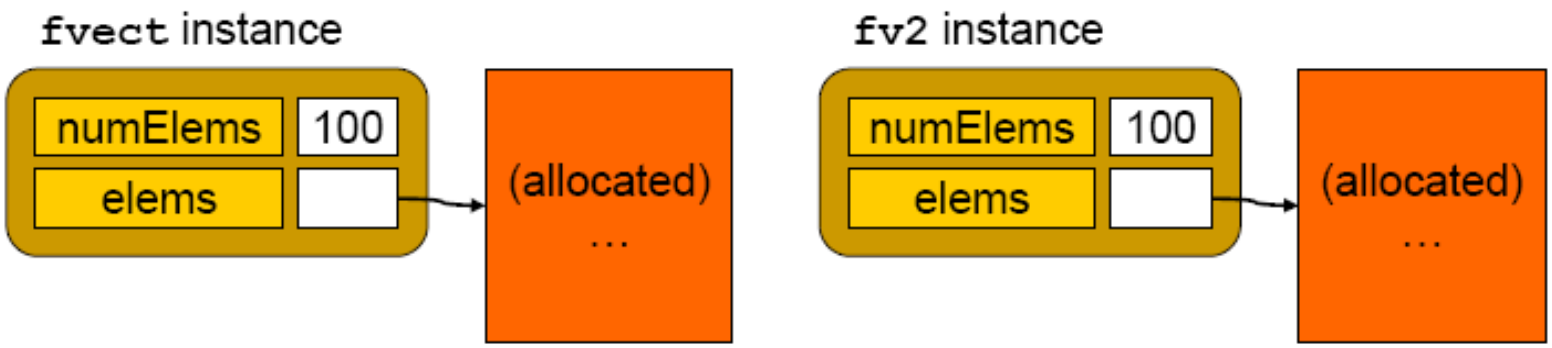
C++ Copy-Konstruktor (4)

- Hier ist der Copy-Konstruktor nicht akzeptabel, da er eine flache Kopie erzeugt

```
FloatVector fv2 = fvect;    // Same as fv2(fvect)
fvect.setelem(3, 5.2);      // Also changes fv2!
```

- Daher muss man seinen eigenen Copy-Konstruktor schreiben:

```
FloatVector::FloatVector(FloatVector &fv) {
  numElems = fv.numElems;
  // DON'T just copy the elems pointer!
  elems = new float[numElems];      // Allocate space
  for (int i = 0; i < numElems; i++) {
    elems[i] = fv.elems[i];         // Copy the data
  }
}
```



C++ Eigenheiten

- C++ setzt bestimmte Klassenoperationen voraus
- Wird etwas nicht gefunden, werden default-Versionen angelegt
- Benötigte Methoden:
  - Ein Copy-Konstruktor
  - Ein Nicht-Copy-Konstruktor
  - Ein Zuweisungsoperator
  - Ein Destruktor
- Aufruf eines Copy-Konstruktors:

```
Point p1(19.6, -3.5);
Point p2(p1);              // Copy p1
```

- Analog (syntaktischer Zucker):

```
Point p1(19.6, -3.5);
Point p2 = p1;             // Identical to p2(p1)
```

- Im Unterschied zu:

```
Point p1(19.6, -3.5), p2;
p2 = p1;
```

Valgrind


Valgrind Home - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://valgrind.org/
Google
Meistbesuchte Seiten Erste Schritte Aktuelle Nachrichten Supremacy 1914 - The... campusnet Wikipedia -EN
Information
About
News
Tool Suite
Supported Platforms
The Developers
Source Code
Current Releases
Release Archive
Front Ends / GUIs
Variants / Patches
Code Repository
Documentation
Table of Contents
Quick Start
FAQ
User Manual
Download Manual
Valgrind
Current release: valgrind-3.3.1
Valgrind is an award-winning instrumentation framework for building dynamic analysis tools. There
Fertig

Valgrind Beispiel

- valgrind programm:

```
==11006== Memcheck, a memory error detector
==11006== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==11006== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==11006== Command: bin/mcubes /home/twiemann/data/kurt3d/flur3.pts 10
==11006==
```

- Fehlermeldungen:

```
==11006== Conditional jump or move depends on uninitialised value(s)
==11006==    at 0x5C8EF12: frexpf (s_frexpf.c:41)
==11006==    by 0x4210D2: StannInterpolator::StannInterpolator(float**, float**, int,
float, int, float, BaseVertex) (cmath:285)
==11006==    by 0x415E68: FastGrid::readPlainASCII(std::string) (FastGrid.cpp:
444)==11006==    by 0x41629C: FastGrid::readPoints(std::string) (FastGrid.cpp:
351)==11006==    by 0x4179C5: FastGrid::FastGrid(Options*) (FastGrid.cpp:73)==11006==
by 0x40AD0B: main (main.cc:40)
```

- Zusammenfassung

```
LEAK SUMMARY:
==11006==    definitely lost: 0 bytes in 0 blocks
==11006==    indirectly lost: 0 bytes in 0 blocks
==11006==      possibly lost: 203,540 bytes in 42 blocks
==11006==    still reachable: 53,247,486 bytes in 584,700 blocks
==11006==         suppressed: 0 bytes in 0 blocks
==11006== Rerun with --leak-check=full to see details of leaked memory
```