Programmieren in C++

SS 2011

Vorlesung 11, Mittwoch 20. Juli 2011

(Konstruktoren II, Type Casting II, Projekt)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem 10. Übungsblatt (Vererbung)
  - Dies ist die **drittletzte** Vorlesung
  - Treffen mit Ihrem Tutor, final call
- Konstruktoren II
  - explizite und implizite Aufrufe bei Vererbung
  - "Initializer" bei Konstruktoren
- Type Casting II
  - static_cast, dynamic_cast, reinterpret_cast
- Übungsblatt 11
  - Entscheiden Sie sich für ein Projekt und schreiben Sie (nach reiflicher Überlegung) alle .h Dateien dazu

Erfahrungen mit dem Ü10 (Vererbung)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 20.7 15:47
  - Schönes Übungsblatt, viel dazu gelernt
  - Wunderbares Übungsblatt, war ca. 2h vor der VL fertig
  - "Die Vermutung liegt nahe, dass Frau Bast möglicherweise sehr gut erklärt hat"
  - Umfang genau richtig (3-4 Stunden im Durchschnitt)
  - Als Beispiel für Vererbung muss man was aus der Geometrie nehmen, heißt es
  - Zufallszahlen selber erzeugen gar nicht so leicht
  - Linter meckert bei Methodenname sort
  - Linter möchte rand_r statt rand wegen thread-safety
  - Vorteil von std::vector gegenüber C-arrays?
  - Unterschied zwischen size_t und int nicht richtig klar

size_t versus int 1/2

UNI FREIBURG

- Warum die Unterscheidung?
  - Ein int kann negativ sein, zum Beispiel -1
  - Ein size_t ist auf jeder Maschine ≥ 0 und kann außerdem größere Werte annehmen als ein int

    32-Bit Maschine: int $-2^{31}..2^{31}-1$ size_t $0..2^{32}-1$

    64-Bit Maschine: int $-2^{31}..2^{31}-1$ size_t $0..2^{64}-1$
  - Deswegen ist es gefährlich, wenn man int in size_t konvertiert, oder die beiden miteinander vergleicht

```
int x = –1;
size_t y = 10000;
if (x < y) ...  // Is this true or false? Compiler warning!
y = x;  // The value of y is 2^32 – 1 or 2^ 64 – 1 now!
```

size_t versus int 2/2

UNI FREIBURG

- Viele STL Methoden geben einen size_t zurück
  - std::vector::size() gibt einen size_t zurück
  - std::string::size() bzw. std::string::length() ebenso
  - std::string::find(...) auch
  - Oder wollen einen als Argument ...
  - An all diesen Stellen benutzt man aus den vorgenannten Gründen deswegen Variablen vom Typ size_t, allen voran bei Iterationen über std::vector oder std::string

```
std::vector<int> A(n);
for (size_t i = 0; i < A.size(); i++) A[i] = rand();
```

Konstruktoren 1/5

UNI FREIBURG

- Implizite Aufrufshierarchie
  - Ohne Weiteres ruft der Konstruktor einer Unterklasse ganz am Anfang erstmal den Konstruktor der Oberklasse auf (der wiederum, bei mehrstufiger Vererbung, den Konstruktor seiner Oberklasse aufruft usw)
  - Ohne Weiteres wird dabei der Default-Konstruktor (= der Konstruktor ohne Argument) aufgerufen
  - Dasselbe dann bei den Destruktoren, nur in umgekehrter Reihenfolge
  - Siehe Code aus letzter Vorlesung, und folgendes Beispiel

Konstruktoren 2/5

UNI FREIBURG

- Implizite Aufrufshierarchie, Beispiel

```
class IndexBase {
  IndexBase() { _fileName = "index.db"; }
  string _fileName;
};

...

class InvertedIndex : public IndexBase {
  InvertedIndex() { }
};

...

InvertedIndex ii;
cout << ii._fileName << endl;  // Prints "index.db".
```

Konstruktoren 3/5

UNI FREIBURG

- Explizite Aufrufe
  - Möchte man einen anderen Konstruktor aufrufen, gibt man das in der Initialisierungsliste des Konstruktors an

```
class InvertedIndex : public IndexBase {
  explicit InvertedIndex(const string& fileName)
    : IndexBase(fileName) {
    ... // Code from the constructor.
  }
};

...
InvertedIndex ii("inv-index.db");
cout << ii._fileName << endl;  // Prints "inv-index.db".
```

Konstruktoren 4/5

UNI FREIBURG

- Explizite Aufrufe
  - Man könnte einen Konstruktor auch explizit aus irgendeiner Methode der Klasse aufrufen

```
void InvertedIndex::someMethod() {
  ...
  InvertedIndex("xxx");  // Calls the constructor (again).
  IndexBase("yyy");  // Calls the constructor of base class.
  ...
}
```

  - Das macht man aber in C++ nicht
  - Falls der Konstruktor Code enthält, den man anderer Stelle gerne nochmal ausführen möchte, sollte dieser Code besser in einer separaten Methode, z.B. reset() oder initialize(), stehen

Konstruktoren 5/5

UNI FREIBURG

- Initializer
  - Allgemeiner kann man in der Liste der sogenannten initializer beliebige Membervariablen initialisieren

```
class InvertedIndex {
 public:
  InvertedIndex(string& indName, string& vocName)
    : IndexBase(indName), _vocabulary(vocName), _size(0)
  { ... }
 private:
  Vocabulary _vocabulary;  // Another class of mine.
  int _size;
};

InvertedIndex ii("inv-index.db");
```

Type Casting 1/5

UNI FREIBURG

- Implizite Konvertierung
  - Wir haben schon gesehen, dass Zeiger auf die Unterklasse bei Bedarf automatisch in Zeiger auf die Oberklasse konvertiert werden

    ```
    Thing* thing = new IntegerThing(4);  // No problem.
    ```

  - Andersrum aber nicht

    ```
    Thing* thing = new IntegerThing(4);
    ...
    IntegerThing* integerThing = thing;  // Will not compile.
    ```

Type Casting 2/5

UNI FREIBURG

- Explizite Konvertierung mit **dynamic_cast**
  - Man kann den Compiler aber dazu zwingen

```
Thing* t = new IntegerThing(4);
IntegerThing* it  = dynamic_cast<IntegerThing*>(t);  // Ok.
```

  - Bei dynamic_cast schaut das Programm **zur Laufzeit**, ob der Zeiger auf ein Objekt vom richtigen Typ zeigt
    - falls ja, wird die Konvertierung durchgeführt
    - falls nein, wird NULL zurückgegeben

```
Thing* t = new CharacterThing(4);
IntegerThing* it  = dynamic_cast<IntegerThing*>(t);
printf("%p\n", it);  // Will print NULL.
```

  - Für ein realistisches Beispiel wo man das braucht, siehe http://ad-svn...cplusplus-ss2010...vorlesung-11/Number.{h, cpp}

Type Casting 3a/5

UNI FREIBURG

- Explizite Konvertierung mit **static_cast**
  - Das schreibt man völlig analog

```
Thing* t = new IntegerThing(4);
IntegerThing* it  = static_cast<IntegerThing>(t);  // Ok.
```

  - Das prüft allerdings nicht nach, ob das sinnvoll ist, sondern kopiert einfach die Zeiger-Adresse
  - Den static_cast benutzt man auch für explizite Umwandlungen zwischen Basistypen

```
double pi = 3.141592653589793;
int piRounded1 = pi;  // Implicit conversion, will compile.
int piRounded2 = static_cast<int>(pi);  // Explicit is better.
int x1 = -1;
unsigned int x2 = static_cast<unsigned int>(x1);  // Dito.
```

Type Casting 3b/5

UNI FREIBURG

- Explizite Konvertierung mit **static_cast**
  - Mit static_cast kann man auch explicit deklarierte ein-argumentige Konstruktoren aufrufen

```
explicit SeqOfIntegers(int x);  // Create seq. with one element.
...
void SeqOfIntegers::append(const SeqOfIntegers& seq);
...
SeqOfIntegers sequence;
sequence.append(5);  // Will not compile, because of explicit.
sequence.append(static_cast<SeqOfIntegers>(5));  // Fine.
```

Type Casting 4/5

UNI FREIBURG

- Explizite Konvertierung mit **reinterpret_cast**
  - Bei static_cast müssen die Klassen immerhin verwandt sein

```
class Xyz {   };  // Class unrelated to Thing;

Thing* t = new IntegerThing(4);
Xyz* xyz  = static_cast<Xyz*>(t);  // Will not compile.
```

  - Mit reinterpret_cast kann man **beliebige** Zeiger-konvertierungen machen, und auch Zeiger ↔ Integer

```
Thing* t = new IntegerThing(4);
Xyz* xyz  = reinterpret_cast<Xyz*>(t);  // Will compile.
size_t x = reinterpret_cast<size_t>(t);  // Will compile.
```

  - Das braucht man allerdings in der Praxis nur ganz selten, bei (sehr) low-level Anwendungen, z.B. Treibersoftware

Type Casting 5/5

UNI FREIBURG

- Wir merken uns
  - Konvertierung Unterklasse* nach Oberklasse* geht automatisch und braucht man praktisch immer, sobald man was mit Vererbung macht
  - Konvertierung andersrum geht mit dynamic_cast und dann auf NULL checken, braucht man aber selten
  - Explizite Typumwandlung über ein-argumentige (explicit) Konstruktoren mit static_cast, kommt öfter mal vor
  - Und reinterpret_cast brauchen wir praktisch nicht

Projekt

UNI FREIBURG

- Sie haben 2 bzw. 3 Projekte zur Auswahl
  - Projekt 1: Suchmaschine
    - Eine Suchmaschine für eine gegebene Menge von Dateien auf der eigenen Festplatte.
  - Projekt 2: Pong
    - Realisierung des Uralt-Video-Spiels mit ASCII-Zeichen auf der Textkonsole
  - Projekt 3: Selbstdefiniert
    - Für die, die in C++ schon sehr sicher / erfahren sind, und für die die Übungsblätter (zu) leicht waren
  - Details dazu auf dem Wiki zur Veranstaltung, insbesondere zu geforderter und optionaler Funktionalität

Projekt

UNI FREIBURG

- Übungsblatt 11
  - Schreiben Sie **alle** .h Dateien **aller** Klassen, die Sie zur Realisierung Ihres Projektes brauchen
  - Sie brauchen (und sollen) noch **nichts** implementieren, wirklich nur die .h Dateien
  - Aber schauen Sie, dass die .h Dateien alle kompilieren (indem Sie alle in einer ...Main.cpp includen, die sonst nichts weiter tut) und den Linter ohne Fehler passieren
  - In der nächsten Vorlesung gebe ich dann für Projekt 1 und 2 diverse Tipps zur Implementierung
  - Außerdem bekommen Sie Feedback von Ihren Tutoren zu Ihren .h Dateien

Projekt 1 — Suchmaschine 1/2

UNI FREIBURG

- Prinzip Vorberechnung
  - In einer Vorberechnung werden die später zu durchsuchenden Dateien einmal alle gelesen und ein sogenannter invertierter Index dazu gebaut
  - Dazu bekommt jede Datei eine Nummer (aufsteigend ab 0 oder 1), und für jedes Wort, das in mindestens einem der Dokumente vorkommt, speichern wir die Liste aller (Nummern von) Dateien, die dieses Wort enthalten, z.B.

    datei1.txt (#1): Hier steht nur Quatsch drin.
    datei2.txt (#2): Hier hier, steht steht, quatsch quatsch.
    datei3.txt (#3): Nur drin steht.

drin:	#1, #3	quatsch:	#1, #2
hier:	#1, #2	steht:	#1, #2, #3
nur:	#1, #3		

Projekt 1 — Suchmaschine 2/2

UNI FREIBURG

- Prinzip Suchanfrage
  - Bei einer Suchanfrage schneidet man einfach die invertierten Listen der Suchbegriffe, und enthält so die Liste der Dateien, die alle Suchbegriff enthalten

    Suchanfrage: nur quatsch

    Invertierte Listen: nur: #1, #3 quatsch: #1, #2

    Schnittmenge: #1
  - Schneiden von sortierten Listen, siehe Vorlesung 9
  - In der Ausgabe will man dann neben dem Dateinamen auch noch die Zeilen, die die Suchbegriffe enthalten, wenn möglich die Suchbegriffe dabei hervorheben

    datei1.txt: Hier steht nur Quatsch drin.

Projekt 2 — Pong 1/2

UNI FREIBURG

- Prinzip
  - Malen auf den Bildschirm einfach mit printf oder cout
  - Aber immer nur das neu malen, was sich verändert hat
  - Dazu muss man den Cursor absolut positionieren können
  - Das geht mit ANSI escape codes, zum Beispiel:

```
printf("\x1b[2J");  // Clears the screen
printf("\x1b[1;1H");  // Put cursor in upper left corner.
printf("\x1b[?25l");  // Hide the cursor
```

  - Kleines Beispielprogramm dazu im SVN
  - Es gibt auch eine Bibliothek dazu, curses bzw. ncurses, die ist sehr mächtig und außerdem portabler als die ANSI codes

```
man 3 ncurses
```

Projekt 2 — Pong 2/2

UNI FREIBURG

- ANSI escape codes vs. ncurses
  - Für einfache Sachen sind die ANSI escape codes einfacher, weil man keinerlei Funktionsaufrufe hat, sondern einfach nur (einfache) Kontrollsequenzen auf den Bildschirm druckt
  - Wenn man, bei der Erfassung von Tastendrücken, verhindern will, dass die entsprechenden Zeichen auf dem Bildschirm erscheinen, wird es ohne ncurses aber schon tricky

```
struct termios term = {0};
tcgetattr(0, &term);
term.c_lflag &= ~ICANON & ~ECHO;
tcsetattr(0, TCSANOW, &term);
```

  - Das geht mit ncurses einfacher und portabler

```
cbreak();
```

Literatur / Links

UNI FREIBURG

- Konstruktoren
  - http://www.cplusplus.com/doc/tutorial/classes/
  - http://www.cplusplus.com/doc/tutorial/inheritance/
- Type Casting
  - http://www.cplusplus.com/doc/tutorial/typecasting/
- Projekte
  - Spezifikation auf dem Wiki zur Veranstaltung

    http://ad-wiki...ProgrammierenCplusplusSS2011/Projekt
  - ANSI escape codes / Curses library

    http://en.wikipedia.org/wiki/ANSI_escape_code

    http://heather.cs.ucdavis.edu/~matloff/.../Curses.pdf