C++ Einführung

Ein kurzer Überblick

University of Siegen • Institute for Vision and Graphics
Computer Graphics and Multimedia Systems Group

- 2 Typen von Dateien:
- Source-File **.cpp**
  - Funktionsdefinitionen (eigentliche Logik / Programm)
- Header-File **.h**
  - Funktions- und Klassendeklarationen (Bauplan)
- meist pro Klasse ein .h/.cpp-Paar
- Verwenden einer Klasse durch Einbinden ihres Headers
  - #include „[Headerdatei]“ eigene Header
  - #include <[Headerdatei]> System- / Framework-Header

- Klassendeklaration in C++ mit Schlüsselwort **class**
- Enthält Deklaration von Daten und Methoden (in C++ members genannt)

Beispiel

```
class Person {
  char* name;
  int age;
  void setName( char* );
  void setAge( int );
};
```

- Unterschiedliche Sichtbarkeit für Teile eines Objekts:
  - **private:** Zugriff nur innerhalb der Klasse
  - **public:** Zugriff von überall
  - **protected:** wie **private**, abgeleitete Klassen haben Zugriff
- **public**-Teil stellt die Schnittstelle für andere Objekte dar
- Standard-Sichtbarkeit ist **private**
- Als **friend** deklarierte Funktionen und Klassen haben **public** Zugriff

Beispiel

```
class Person {
public:
  void setName( char* );
  void setAge( int );

private:
  char* name;
  int age;
};
```

Statische Erzeugung

```
Person peter;
```

- Objekt wird gelöscht, wenn der Scope des Bezeichners verlassen wird

Dynamische Erzeugung

```
Person* peter;
peter = new Person; // Objekt wird erzeugt
delete peter; // Objekt wird gelöscht
```

- Objekt muss explizit gelöscht werden

- Zwei Operatoren in C++:
- Speicherreservierung mit **new**

```
type *pointer_to_type;
pointer_to_type = new type;
```

- Falls Reservierung fehlschlägt, wird eine **std::bad_alloc** Exception geworfen (oder ein **NULL**-Zeiger zurückgegeben)
- Speicherfreigabe mit **delete**

```
delete pointer_to_type;
```

- Keine Garbage Collection in C++,
  Programmierer verantwortlich für die Freigabe
- Auf Zeiger kann auch nach Freigabe noch zugegriffen werden
- **delete** für einen **NULL**-Zeiger ist erlaubt
- Arrays variabler Größe nur mit **new** möglich
- Spezielle Syntax für Arrays:

```
int *ap = new int[7];
delete[] ap;
```

- vgl. C: Speicherverwaltung mit **malloc** und **free**
- Niemals **malloc / free** mit **new / delete** mischen
- new sollte immer mit delete zusammenpassen**:**
  new type / delete pointer_to_type und
  new type[] / delete[] pointer_to_type

Konstruktor

- Automatische Initialisierung des neuen Objekts nach Erzeugung
- Konstruktor muss Objekt in einen konsistenten Zustand bringen
- Definiert als Methode der Klasse
- Name der Methode ist identisch zum Namen der Klasse
- Kein Rückgabewert (nicht einmal **void**)
- Mehrere Konstruktoren durch Überladen
- Deklaration normalerweise im **public**-Teil der Klasse
- Compiler erzeugt einen minimalen Default-Konstruktor (ohne Argumente), falls keiner in der Klasse deklariert wurde

Desktruktor

- Aufräumen vor dem Löschen des Objekts
- Ähnlich zu **finalize** in Java
- Deklariert als Klassenmethode
- Methodenname ist der Klassenname mit **~** davor, meist **virtual**
- Kein Rückgabetyp (nicht einmal **void**)
- Nur ein Destruktor möglich
- Destruktoren haben keine Parameter
- Deklaration normalerweise im **public**-Teil der Klasse
- Compiler erzeugt Default-Destruktor (tut nichts), falls in der Klasse keiner deklariert

- Wann wird der Kopier-Konstruktor verwendet?
  - Objekt ist ein Parameter bei einem Funktionsaufruf (als call-by-value)
  - Objekt ist ein Rückgabewert einer Funktion
  - Initialisierung eines Objekts mit einem existierenden Objekt
- Wichtig: Referenz-Operator **&** verwenden
- Default-Kopier-Konstruktor (vom Compiler erzeugt) kopiert bit für bit

Beispiel

```
Person::Person(const Person& p) {
  name = p.name;
  age = p.age;
}
```

Referenz-Parameter

- Adress-Operator & in Variablendeklaration

```
type& reference_variable = variable_of_type;
```

- Keine eigenständigen Variablen
- Proxy oder Alias für eine andere Variable
- Muss bei Deklaration initialisiert werden
- Operationen auf Referenzvariablen verändern die referenzierte Variable
- Ähnlich zu Zeigern mit impliziter Dereferenzierung

- Gleicher Funktionsname für unterschiedliche Implementierungen
- Überladene Funktionen werden unterschieden durch:
  - Anzahl der Parameter
  - Typ der Parameter
  - Reihenfolge der Parametertypen
  - Nicht: Rückgabe-Typ (Rückgabe-Wert kann ignoriert werden)

Beispiel

```
void print();
void print(int, char*); // okay
int print(float); // okay
int print(); // Fehler, nicht unterscheidbar
```

- Funktionsparameter können einen Defaultwert besitzen
- Wird verwendet, wenn der Parameter im Aufruf fehlt
- Nur am Ende der Parameterliste erlaubt

Beispiel

```
void print(char* string, int nl = 1);
print("Test", 0);
print("Test"); // äquivalent zu print( "Test", 1 )
print(); // falsch, Paramter char* fehlt
```

Statische Member-Variablen

- Normalerweise hat jedes Objekt seine eigene Menge an Variablen
- Ausnahme: Member-Variablen, die als **static** deklariert sind
- Elemente, die als **static** deklariert werden, exisieren genau einmal für jede Klasse, egal wie viele Objekte dieser Klasse es gibt
- Ermöglicht es, eine gemeinsame Variable für alle Instanzen einer Klasse zu verwenden
- Zugriffsrechte können wie bei Instanzvariablen festgelegt werden

- Globale Initialisierung außerhalb der Klasse (cpp-File)
- Methoden, die nur auf **static**-Elemente zugreifen, können ebenfalls als **static** deklariert werden
- **static**-Methoden können ohne Objekt aufgerufen werden
- Kein **this**-Zeiger, kein Zugriff auf Instanzvariablen/-methoden der Klasse

- Zweck: Verwendung existierender Implementierungen (Klassen)
- Neue Klasse erbt Eigenschaften einer existierenden Klasse
- Mehrfachvererbung möglich (anders als in Java)
- Syntax:

```
class subClass :
[modifier] superClass1, [modifier] superClass2, ... {
  Deklaration von neuen Member-Variablen und
  neuer oder überschriebener Member-Funktionen (Methoden)
}
```

- Generell: Alles was nicht überschrieben wird, wird geerbt
- Nicht vererbt werden:
  - Konstruktoren
  - Destruktor
  - Zuweisungsoperator

- Konstruktor der abgeleiteten Klasse ruft Konstruktoren der Basisklassen via Initialisierungsliste auf

```
class_name::class_name(parameter_list)
: superclass1(parameters), superclass2(parameters), ...
```

- Destruktor der Basisklasse wird automatisch aufgerufen nach dem Destruktor der abgeleiteten Klasse

Virtuelle Methoden

- Interface wird durch Typ eines Zeigers festgelegt (statischer Typ), nicht durch Typ eines Objekts, auf das der Zeiger zeigt (dynamischer Typ)
- Zugriff auf alle Elemente einer abgeleiteten Klasse nur nach einem Casting des Zeigers

- Ziel: Ausführung einer geeigneten Methode der Unterklasse, ohne diese explizit zu kennen
- Lösung: Virtuelle Methoden
- Schlüsselwort **virtual** in der Basisklasse
- Aufgerufene Methode wird zur Laufzeit bestimmt
- Compiler erzeugt vtables (Sprungtabellen für virtuelle Methoden)

- Nicht alle deklarierten Methoden werden auch implementiert
- Es kann keine Instanzen/Objekte dieser Klasse geben
- Unterklassen können nur dann Instanzen haben, wenn alle deklarierten Methoden implementiert werden
- Abstrakte Klassen können verwendet werden
  - als Basisklassen ohne Instanzen (**class** mit **abstract**-Methoden in Java)
  - zur Definition eines Typs/einer Schnittstelle (**interface** in Java)

- Syntax für nicht implementierte Methoden (rein virtuell):

```
class class_name {
  virtual  return_type  method_name( parameter_list ) = 0;
};
```

- Zeiger auf abstrakte Klassen möglich, müssen aber mit Objekten einer nicht abstrakten Unterklasse initialisiert werden

- **reinterpret_cast**:
  - Zeigertyp in einen beliebigen anderen Zeigertyp umwandeln
  - Der Wert des Zeigers wird dabei nicht verändert
  - Keinerlei Überprüfungen

- **static_cast**:
  - Nur alle implizit möglichen Casts und deren Umkehrungen unterstützt
  - Bsp: int <-> char, Zeiger auf Basisklasse in Zeiger auf abgel. Klasse
  - Keine echte Überprüfung auf Typkompatibilität

- **dynamic_cast**:
  - Verwendet Run-Time Type Information, um zu überprüfen, ob der Cast möglich ist
  - Entspricht dem Casting in Java
  - Liefert NULL zurück falls nicht möglich, keine Exception geworfen
- **const_cast**:
  - Änderung der **const**-Eigenschaft (nur hiermit möglich)

- **struct**-Schlüsselwort bietet Alternative zu Klassen um Datentypen zu einem neuen Datentyp zu vereinen
  - Zugriff stets **public**

```
struct Point {
  int x;
  int y;
};
```

Strings werden von C++ nur als Array von Buchstaben unterstützt

```
const char* string = „Hello World.“
```

Standard Template Library (STL) bietet String-Klasse ähnlich dem Java String-Types

Außerdem: z.B. dynamische Arrays, Listen, Maps, Heaps

Eingabe und Ausgabe zu Strömen (Streams) via Operatoren

- **std::cin** Eingabestrom
- **std::cout** Ausgabeströme (cerr, clog)
- **>>** Eingabe-Operator
- **<<** Ausgabe-Operator

Beispiel

```
#include <iostream>
using namespace std;

void main() {
  int test;
  cin >> test;
  cout << "test=" << test << endl;
}
```