Programmieren in C++

SS 2011

Vorlesung 5, Mittwoch 1. Juni 2011

(Klassen, Objekte, Methoden, new & delete)

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 4. Übungsblatt
- Themen heute
  - Erste objektorientierte Schritte
  - Dynamische Speicherallokation mit new und delete
  - Übungsblatt dazu:
    - Code vom letzten Übungsblatt objektorientiert machen
    - und mit dynamischer Speicherallokation

Erfahrungen mit dem Ü4 (Zeiger etc.)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 1.6 16:10
  - Die meisten fanden es einfacher als das Ü3 (Mandelbrot)
  - Zeiger interessant, aber noch nicht ganz verstanden
  - Auswahl einfachere / schwerere Aufgabe war gut
  - Nervig alle Eingabefehler abzufangen
  - Darf man sowas wie isdigit aus ctype.h benutzen?
  - Linter ist lästig, jeder hat doch seinen Stil
  - Tests waren sehr hilfreich
  - printf versus cout, etc.
  - Videoaufzeichnungen sind ziemlich groß
  - Gespannt auf die Musterlösung für die 1b

Klassen

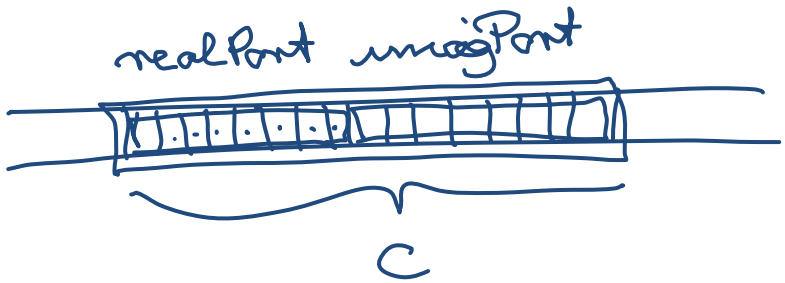
UNI FREIBURG

- Für uns erstmal zwei Verwendungszwecke
  - Definition von neuen (komplexeren) Datentypen
    - durch die **Membervariablen** einer Klasse
  - Zusammenfassen von Funktionen die "zusammen gehören"
    - das sind die **Methoden** einer Klasse

Membervariablen einer Klasse

- Deklaration der Klasse
  - Man schreibt einfach alle Variablen, die zu dem Datentyp gehören sollen in die Klasse, wie bei einer Deklaration

```
class ComplexNumber
{
   double realPart;
   double imaginaryPart;
};
```



- Deklaration eines Objektes der Klasse
  - Analog zur Deklaration für die Basisdatentypen:

```
ComplexNumber c;
```

  - **Wichtig:** Das reserviert, **anders als bei Java**, bereits Speicherplatz, und c ist der Name für dieses Stück Speicher

Methoden eine Klasse

UNI FREIBURG

- Zu der Klasse gehörige Funktionen heißen Methoden
  - Sie werden innerhalb der Klasse (in der .h Datei) deklariert

```
class ComplexNumber
{
   double computeNorm();
   double realPart;
   double imagPart;
};
```

  - Implementierung in der .cpp Datei; man beachte, dass zum vollen Methodennamen der Klassenname gehört

```
 double ComplexNumber::computeNorm()
{
   return sqrt(realPart * realpart + imagPart * imagPart);
}
```

Zugriff auf Membervariablen & Methoden

UNI FREIBURG

- Das geht über den **.** Operator

```
ComplexNumber c;
printf("Real part: %f\n", c.realPart);
printf("Imaginary part: %f\n", c.imagPart);
printf("The norm is: %f\n", c.computeNorm());
```

  man beachte den Unterschied zu `computeNorm(c)`

- Zugriffsberechtigung
  - Damit das kompiliert, muss die Funktion / Methode, in der dieser Code steht auch **Zugriffsberechtigung** für diese Membervariablen haben → nächste Folie

Zugriffberechtigung

UNI FREIBURG

- Public und Private
  - Vor den Abschnitt mit den Membervariablen und Methoden, auf die von außen (über ein Objekt) zugegriffen werden soll schreibt man (nur ein Leerzeichen eingerückt)

    public:
  - Vor den Abschnitt mit den Membervariablen und Methoden, auf die von außen (über ein Objekt) nicht zugegriffen werden soll, schreibt man (nur ein Leerzeichen eingerückt)

    private:
  - Es gibt auch noch protected: das lernen wir erst später
  - In den Methoden einer Klasse kann man grundsätzlich auf alle Membervariablen und Methoden dieser Klasse (egal von welchen Objekten) zugreifen

Benennung von Membervariablen 1/2

UNI FREIBURG

- Wie unterscheidet man Membervariablen von normalen Variablen?

```
double ComplexNumber::someStupidMethod()
{
  double realPart = 2 * realPart;
  printf("%f\n", realPart);  // What is printed now?
};
```

  - Bei Mehrdeutigkeiten nimmt der Compiler die zuletzt deklarierte Variable
  - Aber besser man vermeidet solche Mehrdeutigkeiten!
  - Dazu versehen wir alle Membervariablen mit einem _

Benennung von Membervariablen 2/2

- Das sieht dann so aus:

```
class ComplexNumber
{
   double computeNorm();
   double _realPart;
   double _imagPart;
};

double ComplexNumber::computeNorm()
{
   return sqrt(_realPart * _realPart + _imagPart * _imagPart);
}
```

  - Das ist nur eine Namenskonvention, sonst nichts!

FRIEND_TEST

UNI FREIBURG

- Membervariablen sind in der Regel private
  - Weil der Zugriff auf die Werte eines Objektes meist über die Methoden der Klasse ist
    - Die Idee dahinter ist, dass ein Benutzer der Klasse nicht wissen muss, wie das Objekt innen aussieht
  - In den Tests will man aber oft Zugriff auf die private Member-variablen, z.B. im TEST(ListOfIntegers, parseFromString)

```
ListOfIntegers list;
list.parseFromString("5");
ASSERT_EQ(5, list._elements[0]);  // _elements is private.
```

  - Dazu schreibt man unter die Klassendeklaration

```
parseFromString(const char* list);
FRIEND_TEST(ListOfIntegers, parseFromString);
```

Konstruktor 1/3

UNI FREIBURG

- Objekte sind ohne Weiteres erstmal uninitialisiert
  - Anders als bei den Basistypen (int, char, double, bool, etc.) ist es allerdings üblich, sie zu initialisieren
  - Das tut man über den sogenannten Konstruktor
  - Der Konstruktor ist eine Methode, die so heißt wie die Klasse und keinen Rückgabewert hat
  - Deklaration innerhalb der Klasse in der .h Datei

```
class ComplexNumber
{
   ComplexNumber(double realPart, double imagPart);
   double _realPart;
   double _imagPart;
};
```

Konstruktor 2/3

- Implementierung und Aufruf
  - Implementierung in der .cpp Datei

```
ComplexNumber::ComplexNumber(double realPart,
                             double imagPart)
{
   _realPart = realPart;
   _imagPart = imagPart;
}
```

  - Aufruf erfolgt automatisch bei der Deklaration eines Objektes

```
ComplexNumber c(0, 0);  // Calls the constructor.
```

Konstruktor 3/3

UNI FREIBURG

- Defaultkonstruktor
  - So heißt der Konstruktor ohne Argumente

    ```
    ComplexNumber c;  // Calls the default constructor.
    ```

  - Den Defaultkonstruktor gibt es auch ohne, dass man ihn explizit deklariert, und er tut ... nichts
  - Man kann ihn aber auch explizit in der .h Datei deklarieren

    ```
    ComplexNumber();
    ```

    ... und in der .cpp Datei implementieren

    ```
    ComplexNumber::ComplexNumber()
    {
       printf("Hello, I am your new default constructor\n");
    }
    ```

Statische Membervariablen

UNI FREIBURG

- Variablen, die zu der Klasse als Ganzes gehören

  … und nicht zu jedem einzelnen Objekt

  – Zum Beispiel unsere MAX_LIST_SIZE
  – Die deklariert man dann in der Klasse so:

  ```
  static const int MAX_LIST_SIZE = 100;
  ```

  – Bei nicht-const Variablen muss die Initialisierung in der .cpp Datei erfolgen

  ```
  static int numObjectsCreated; // In the class declaration.
  ...
  ListOfIntegers::numObjectsCreated = 0; // In the .cpp file.
  ```

Dynamische Speicherallokation 1/3

UNI FREIBURG

- Statische Speicherallokation
  - Bei der Deklaration von einem Feld muss die Größe des Feldes bekannt sein

```
int n = atoi(argv[1]);
int array[n];  // This will not compile, n must be const.
```

- Dynamische Speicherallokation
  - Zur Laufzeit bekommt man Speicher mit **new**

```
int n = atoi(argv[1]);
int* array = new int[n];
```

  - Das new alloziert in dem Fall Speicherplatz für n ints und gibt einen Zeiger auf den Anfang dieses Platzes zurück

Dynamische Speicherallokation 2/3

UNI FREIBURG

- Dynamische Allokation von einzelnen Objekten
  - Geht analog

    ```
    int* p1 = new int;  // Allocate space for a single int.
    ```

  - Für Klassen wird dabei der Konstruktor aufgerufen

    ```
    int* p2 = new ComplexNumber(0, 0);
    ```

  - Ohne Argumente der Defaultkonstruktor

    ```
    int* p3 = new ComplexNumber();
    ```

  - Ohne Klammern geht hier nicht

    ```
    int* p4 = new ComplexNumber;  // Will not compile!
    ```

Dynamische Speicherallokation 3/3

UNI FREIBURG

- Freigeben von Speicher
  - Dynamisch allozierten Speicher sollte man freigeben, wenn man ihn nicht mehr braucht, das geht mit **delete**

```
int n = atoi(argv[1]);
int* array = new int[n];  // Pointer to an array of n ints.
int *p1 = new int;  // Pointer to single int.
int* p2 = new ComplexNumber();  // Pointer to single object.
... some code ...
delete[] array;  // For pointers to arrays use delete[].
delete p1;  // For pointers to single objects use delete.
delete p2;  // Dito.
```

Destruktor 1/2

UNI FREIBURG

- Der Destruktor wird aufgerufen wenn

  ... das Objekt seinen sogenannten **scope** verlässt

  – Der scope einer Variablen oder eines Objektes beginnt bei der letzten **{** davor und endet an der dazugehörigen **}**

```
void someFunction()
{
   // A new scope has just begun.
   ...
   ListOfIntegers list;  // Constructor called.
   ...
   // Scope ends now, destructor of list will be called.
}
```

Destruktor 2/2

UNI FREIBURG

- Der Destruktor ist auch einfach eine Methode
  - Und heißt wie der Konstruktur, nur mit einer **~** davor
  - In der Deklaration der Klasse in der .h Datei:

    ```
    ~ListOfIntegers();
    ```

  - Implementierung in der .cpp Datei:

    ```
    ListOfIntegers::~ListOfIntegers()
    {
       ...
    }
    ```

  - Wie beim Konstruktor muss man nicht explizit einen Destruktor deklarieren / implementieren
  - Es wird dann am Lebensende des Objektes der **Default-Destruktor** aufgerufen ... der einfach nichts tut

Operatoren

- Es wäre ja schön, wenn man schreiben könnte

```
ListOfIntegers list;
list.parseFromString("1 2 3");
printf("%d\n", list[1]);  // Access like it were a normal array.
```

  - Das geht, indem man für die betreffende Klasse eine Methode mit dem Namen **operator[]** implementiert

```
int ListOfIntegers::operator[](int i)
{
   return _elements[i];
}
```

  - Aufruf geht dann so

```
printf("%d\n", list.operator[](1));  // Possible, but not nice.
printf("%d\n", list[1]);  // Same result and looking nice.
```

Literatur / Links

- Klassen, Objekte, Methoden, Konstr. und Destr.
  - http://www.cplusplus.com/doc/tutorial/classes/
- Operatoren
  - http://www.cplusplus.com/doc/tutorial/classes2/
- Dynamische Speicherallokation mit new & delete
  - http://www.cplusplus.com/doc/tutorial/dynamic/
- FRIEND_TEST
  - http://code.google.com/p/googletest/wiki/AdvancedGuide#Testing_Private_Code