

Computer Vision
and Geometry Lab

Informatik I for D-MAVT

Exercise Session 9

Organisatorisches

- Teaching Assistant
 - Alexander Schwing(aschwing@inf.ethz.ch)
 - CAB G 89
 - Website:
<http://www.alexander-schwing.de/Info1DMaVt/>
- Übungsabgabe
 - Auf Papier, keine Emails
 - Später wenn Programme umfangreicher werden, ev. auch Abgabe per Email möglich
- Testatbedingungen
 - 75% aller Übungen
 - Voraussichtlich 12 Serien (d.h. 9 gelöste Serien)

Themen

- Allgemeine Bemerkungen
- Nachbesprechung
- Repetition
- Vorberechnung Serie 9

Bemerkungen

- Selbständig sinnvolle Lösung zu erarbeiten ist wichtige Komponente des Programmierers
 - Es gibt nur selten ‘Kochrezepte’
 - Tipp: Lösung zusammensetzen aus Lösungen von einfacheren Teilproblemen
 - Bsp: Wieso programmieren wir zuerst eine `findMax` Funktion?
 - um sie in `MaxSort` zu verwenden!
- Antwort auf Probleme zuerst selber suchen
 - Übungslides, Vorlesungsunterlagen, Aufgabenblatt
 - Internet
 - Falls keine Antwort in nützlicher Zeit gefunden wird, helfe ich gerne weiter

Nachbesprechung

- gutes Beispiel für MaxSort:

- ```
void maxSort(double* arr, int n) {
 if (n > 1) {
 int m = max(arr, n);
 double tmp = arr[m];
 arr[m] = arr[n-1];
 arr[n-1] = tmp;
 maxSort(arr, n-1);
 }
}
```

- Nachteil

- Rekursion ist meist langsamer als Lösung mittels Schleife
- Wieso?
  - Overhead um Funktion aufzurufen

# Nachbesprechung

- Variablen initialisieren
- Was soll folgende Abfrage beim Bestimmen des maximalen Elements bewirken?
  - ```
if (arr[i] < arr[i+1] && arr[max] < arr[i+1])  
    { ... }
```
- Variablen initialisieren
- Auf ein `new` muss **zwingend** später einmal ein `delete` folgen
 - Für ein dynamisches Array: `delete []`
- Variablen initialisieren
- Vermeiden von globalen Variablen
 - Code kann einfacher wiederverwendet werden ohne globale Variablen

Repetition: Was bisher geschah.

- Typen & Variablen
 - Native Typen: int, float, bool, ...
 - Eigene Typen: Structs
- Expressions (Boolean, mathematische, etc.)
- Kontrollstrukturen
 - if Statement
 - Schleifen (for, while, do-while)
 - Switch-case
- Pointer, Referenzen
- Einfache Datenstrukturen, z.B.
 - FIFO, LIFO
 - verkettete Listen
 - Stack, Trees

Repetition: Was bisher geschah.

- Variablen++
 - Access scope: global vs. lokal
 - Speicherort: Stack vs. Heap
 - Beispiel: statische und dynamische Arrays
- Speicher (what's going on under the hood?)
 - Dynamisch
 - Explizit mittels `new` und `delete` (resp. `delete []`)
 - Statisch
 - Automatisiert vom Compiler
- Funktionen
 - Call-by-value
 - Call-by-reference
 - Stackframe
 - Rekursion
 - Overloading

Repetition: Was bisher geschah.

- Lediglich die Basics
 - Sogenanntes prozedurales Programmieren
- Ab jetzt: Advanced Programming
 - Classes
 - Object Oriented Programming (OO Programming)

OO Programmieren: Definitionen

- Klasse
 - Abstrakte Beschreibung eines Datentyps (Daten + Methoden die aufgerufen werden können)
 - Klassen definieren Typen
- Objekt
 - Eine Variable, deren Datentyp eine Klasse ist
 - Man sagt auch, ein Objekt ist eine Instanz einer bestimmten Klasse
- Methoden
 - Funktionen, um mit einem Objekt zu interagieren (Daten abfragen, ändern, ...)
- Attribute
 - Die eigentlichen Daten

Classes

- Blaupausen für Objects (aka. Instances) dieser Klasse

- Eine Klasse definiert

- die möglichen Zustände und
- das Verhalten / die Interaktionsmöglichkeiten

der Objekte dieser Klasse.

- Beispiel:

```
class Dog {
    public:
        int age;
        float weight;
        void eat(float amountOfMeat);
        void sleep(float duration);
        void playWith(Dog& otherDog);
};

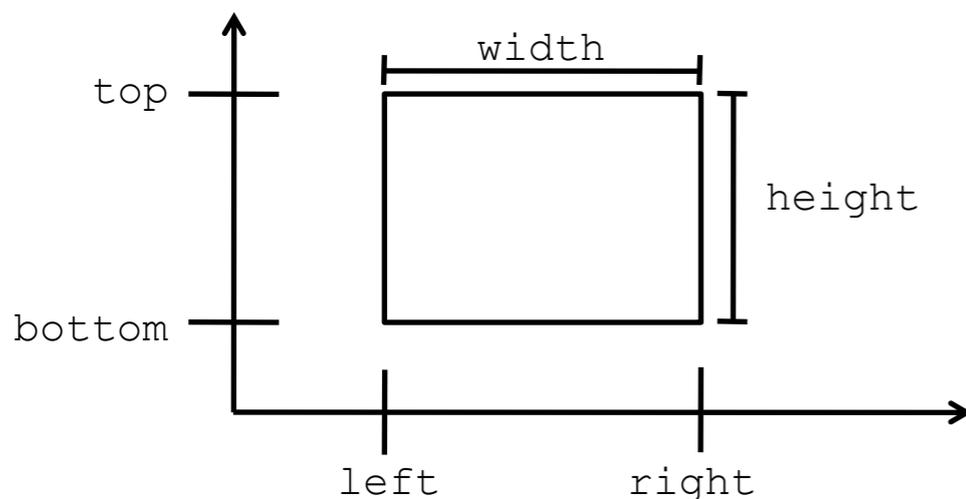
void Dog::eat(float amountOfMeat) {
    weight += amountOfMeat;
    if (amountOfMeat > 0.5)
        sleep(2.0);
}

...
```

```
void main(int argc, char** argv) {
    Dog therry, susi;
    therry.age = 4;
    therry.weight = 12.5;
    ...
    therry.eat(0.6);
    susi.playWith(therry);
    ...
}
```

Classes

- Klassen sind abstrakte Datentypen
 - Es ist für den Benutzer der Klasse nicht wichtig, **wie** die Zustände gespeichert werden
 - Wichtig ist, wie man mit dem Object interagieren kann
- Beispiel: Fläche eines achsen-alinierten Rechtecks



```
class AxisAlignedRectangle {
    float left;
    float right;
    float top;
    float bottom;
    float getArea() {
        return (right - left)*(top-bottom);
    }
};
```

- 4 Werte definieren ein achsen-aliniertes Rechteck
- Alle verbleibenden Werte können aus den 4 gegebenen berechnet werden
 - Effektive Berechnung ist für Benutzer nicht von Interesse
 - Benutzer muss nur wissen, wie er mit Rechteck interagieren kann

```
class AxisAlignedRectangle {
    float top;
    float left;
    float width;
    float height;
    float getArea() {
        return width*height;
    }
};
```

```
class AxisAlignedRectangle {
    float top;
    float left;
    float width;
    float area;
    float getArea() {
        return area;
    }
    float getHeight() {
        return area / width;
    }
};
```

Classes

- Interaktionsmöglichkeiten werden definiert mittels Methoden
 - Eine Methode hat Zugriff auf alle Variablen der Klasse (siehe auch Access Specifier)
 - Jedes Object/Instanz besitzt 'seine eigenen' Variablen um seinen Zustand zu speichern
 - Analog zu structs:
 - ```
struct Point{
 float x, y;
};
Point p1 = {0.1, 0.2}; // p1 besitzt seine eigenen Instanzen der Variablen x und y
Point p2 = {0.3, 0.4}; // p2 besitzt seine eigenen Instanzen der Variablen x und y
```
- Konzept der Datenkapselung (Data Hiding)
  - Lediglich Interface (Interaktionsmöglichkeiten, meist Methoden) muss bekannt sein
  - Von aussen gesehen sind Daten versteckt
    - Lediglich Object selbst sollte direkten Zugriff auf seine Variablen haben

# Classes vs. Structs

- Klasse = Struct + 'eigene' Methoden
  - Structs können eigentlich auch Methoden enthalten
  - Unterschied zwischen Klasse und Struct besteht lediglich im default access specifier:
    - Struct: default ist `public`
    - Class: default ist `private`

# Access Specifier

- Definiert die Zugriffsrechte auf
  - Methoden
  - Attribute
- Für Klassen standardmässig
  - `private`
    - Nur Objekt selbst hat Zugriff auf diese Methoden / Attribute
- Für Structs standardmässig
  - `public`
    - Auch andere ('fremde') Objekte haben Zugriff

# Pointers auf Object

- Klassen sind Datentypen
  - Also können auch Pointer auf Instanzen einer Klasse definiert werden

- Beispiel

```
■ class MyClass {
 public:
 void getName() {
 std::cout << "What is your name?" << std::endl;
 std::cin >> name;
 }
 void sayHello() {
 std::cout << "Hello " << name << std::endl;
 }
 private:
 string name;
};

...
MyClass* ptr = new MyClass();
ptr->getName();
ptr->sayHello();
delete ptr;
```

# Prototyp und eigentlich Implementierung

- Erfolgt Implementierung einer Methode innerhalb der Klassendefinition, dann wird sie [falls möglich] als inline-Funktion angeschaut
- Implementierung der Methoden kann auch ausserhalb der Klassendefinition erfolgen
  - Innerhalb der Klasse steht dann lediglich der Prototyp
  - Klassenname und Scope Operator nötig

# Prototyp und eigentlich Implementierung

- In Datei MyClass.h

```
#include <iostream>
#include <string>
class MyClass {
public:
 void getName();
 void sayHello() { std::cout << "Hello " << name << std::endl; }
private:
 string name;
};
```

- In Datei MyClass.cpp

```
#include "MyClass.h"
void MyClass::getName() {
 std::cout << "What is your name?" << std::endl;
 std::cin >> name;
}
```

- In Datei main.cpp

```
#include "MyClass.h"
int main(int argc, char** argv) {
 MyClass* ptr = new MyClass();
 ptr->getName();
 ptr->sayHello();
 delete ptr;
 return 0;
}
```

# This

- Keyword `this`

- Vordefinierter Zeiger auf das Objekt, auf welchem die Methode aufgerufen wurde

- Beispiel:

```
class Rect{
 public:
 setValues(int x, int y) {
 this->x = x;
 this->y = y;
 }
 private:
 int x, y;
};
```

# Konstruktor

- Wie soll der Zustand eines Objekt initialisiert werden?
  - `MyClass myClass;`  
`MyClass.setX(x);`  
`MyClass.setY(y);`  
...
  - Etwas mühsam...
- Lösung: Konstruktor
  - Wird beim Instanzieren (d.h. beim Entstehen) eines Objekts aufgerufen

# Konstruktor

- Es koennen mehrere Konstruktoren definiert werden
  - Müssen sich allerdings in Inputargumenten unterscheiden
  - Analog zu Function Overloading
- Default C'tor
  - Besitzt gar keine Inputargumente
- Compiler generiert automatisch 2 Konstruktoren + 1 Operator (alle `public`)
  - Default C'tor: `MyClass::MyClass()`
    - 'Selbstinitialisierung' in Default State
  - Copy C'tor: `MyClass::MyClass(const MyClass& )`
    - Initialisierung von einer anderen Instanz derselben Klasse
  - Assignment Operator: `MyClass& MyClass::operator=(const MyClass&)`
    - Kopiert Zustand einer anderen Instanz derselben Klasse
    - Beide involvierten Objekte wurden bereits initialisiert!

# Default C'tor

- Achtung: Sobald ein eigener C'tor definiert wird, generiert der Compiler keinen Default C'tor mehr!

- ```
class MyClass {  
    public: int a,b,c;  
    MyClass(int n, int m) { a=n; b=m; }; // <- Self-defined C'tor present  
    void multiply () { c=a*b; };  
};
```

```
MyClass c1(2,3); // correct
```

```
MyClass c2;      // wrong, there is no default C'tor anymore!
```

Default C'tor

- Was ist hier falsch?

- `MyClass c1;`

- `MyClass c2 ();`

- Compiler fasst letzteres als Deklaration einer Funktion `c2` auf, die keine Inputargumente besitzt und ein Object vom Typ `MyClass` zurück gibt.

Copy C'tor

- Kopiert alle Daten eines anderen Objekts (desselben Typs) in die eigenen Attribute
 - Beispiel:

```
MyClass::MyClass(const MyClass& rv) {  
    a=rv.a; b=rv.b; c=rv.c;  
}  
...  
MyClass c1(2,3);  
MyClass c2(c1); // <- Calls copy-c'tor to initialize c2
```

Konstruktor: Constant and Reference Variables

- ```
class MyClass {
 public:
 const int ci; // <- Constant variable
 int& ri; // <- Reference variable
};

...
MyClass c; // Compiler Error!
```
- Folgender C'tor löst Problem auch nicht:
  - ```
MyClass::MyClass(int _ci, int* pi) {  
    ci = _ci;  
    ri = *pi;  
}
```
- Referenzen und Konstanten müssen zum Definitionszeitpunkt initialisiert werden.
 - Spezielle Syntax:

```
MyClass::MyClass(int _ci, int* pi) : ci(_ci), ri(*pi) { ... }
```
 - Wichtig auch zum Initialisieren der Parents (siehe später bei Vererbung)

Destruktor

- Wird automatisch aufgerufen, wenn das Objekt zerstört wird
 - Gründe fuer Zerstörung
 - Objekt ist eine lokale Variable (d.h. auf Stack) und sein Scope wird verlassen
 - Objekt ist eine dynamische Variable und `delete` wird aufgerufen
- Es ist lediglich ein Destruktor möglich (im Gegensatz zu C'tors)
 - Keine Inputparameter
 - Keinen Rückgabewert
- Zweck des Destruktors: Rückgabe von zuvor verlangten Ressourcen
 - Dynamischer Speicher
 - File Handles
 - ...
- Beispiel:
 - ```
class MyClass {
 public:
 MyClass(int sz) { arr = new int[sz]; } // Constructor
 ~MyClass() {delete[] arr; arr = NULL; } // Destructor
 private:
 int* arr;
};
```

# Wiederholung: Dynamische Speicherallokation (siehe Session 6)

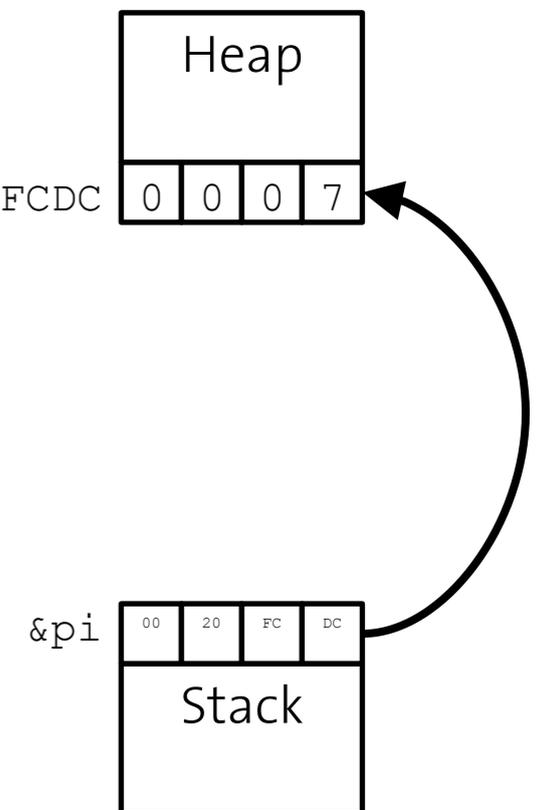
## ■ Operator new

1. Reserviert Speicherbereich für neue Variable (Standardmässig im Heap)
2. Ruft einen Constructor der Variablen auf
3. Gibt einen Pointer auf den neuen Speicherbereich zurück

## ■ Beispiel:

```
int* pi = new int; // Aufrufen des Default-Constructors
int* pi = new int(7); // Aufrufen des Copy-Constructors
```

- Speicher für Integer Pointer `pi` wird im Stack reserviert `pi == 0020FCDC`
- Speicher für den vom Pointer referenzierten Integer jedoch wird im Heap reserviert
- Pointer stellt somit Verbindung zwischen Variablen auf Stack und free store her



# Wiederholung: Dynamische Speicherallokation (siehe Session 6)

- Operator `delete`
  1. Aufrufen des Destruktors des zu löschenden Objects
  2. Freigeben des zuvor mittels `new` reservierten Speicherbereichs
- Funktioniert auch für Arrays (mit angepasster Syntax):
  - `// n muss nicht zur Compilezeit bekannt sein!`  
`int* ai = new int[n];`  
`...`  
`delete[] ai;`
  - `delete[]` ruft den Destruktor von jedem Arrayelement auf und gibt zuletzt den Speicher frei

# Low-Level Details (Advanced)

- `new` und `delete` sind Operatoren

- Definiert im Headerfile `<new>`
- Definiert jeweils 3 Versionen für `new`, `delete`, `new[]`, `delete[]`
- Die zwei wichtigsten Versionen sind:

```
void* operator new (std::size_t size) throw (std::bad_alloc);
```

- “allocates *size* bytes of storage space, aligned to represent an object of that size, and returns a non-null pointer to the first byte of this block. On failure, it throws a [bad\\_alloc](http://en.cppreference.com/errno/bad_alloc) exception.” [<http://www.cplusplus.com/reference/std/new/operator%20new/>]

```
void operator delete (void* ptr) throw ();
```

- “deallocate the memory block pointed by *ptr* (if not-null), releasing the storage space previously allocated to it by a call to [operator new](http://en.cppreference.com/errno/operator_new) and making that pointer location invalid.” [<http://www.cplusplus.com/reference/std/new/operator%20delete/>]

# Low-Level Details (Advanced)

- `void* operator new (std::size_t size) throw (std::bad_alloc);`
  - “operator new can be called explicitly as a regular function, but in C++, new is an operator with a very specific behavior: An expression with the new operator, first calls function operator new with the size of its type specifier as first argument, and if this is successful, it then automatically initializes or constructs the object (if needed). Finally, the expression evaluates as a pointer to the appropriate type.”  
[<http://www.cplusplus.com/reference/std/new/operator%20new/>]
  - Beispiel:  
`MyClass* p = (MyClass*) operator new (sizeof(MyClass));`
    - Dadurch wird aber Constructor von MyClass **nicht** aufgerufen!
- `void operator delete (void* ptr) throw ();`
  - “operator delete can be called explicitly as a regular function, but in C++, delete is an operator with a very specific behavior: An expression with the delete operator, first calls the appropriate destructor (if needed), and then calls function operator delete to release the storage.”  
[<http://www.cplusplus.com/reference/std/new/operator%20delete/>]

# Vorbesprechung Serie 9

- Serie hat nichts mit Classes zu tun
  - Die nächste Serie aber wahrscheinlich schon...
- Rekursion

# Vorbesprechung Serie 9: Aufgabe 1

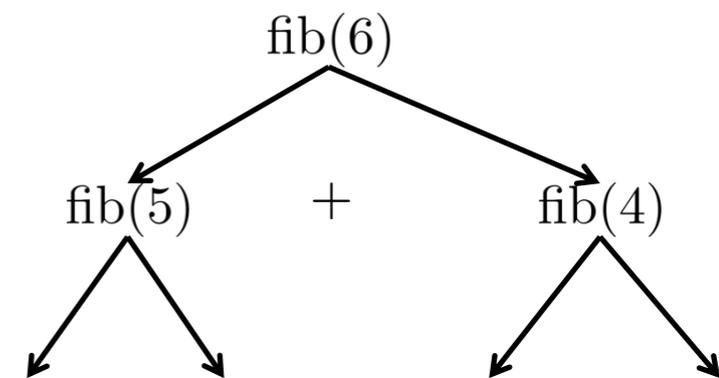
## ■ Rekursion

### ■ Einmal mehr Fibonacci-Folge:

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2) \text{ for } n \geq 2$$



### ■ Teilaufgabe a)

- Auf Papier Rekursion (inkl. aller Zwischenschritte) aufzeichnen

### ■ Teilaufgabe b)

- Wieso ist diese Rekursion ineffizient?  
Verbesserungsvorschläge?



# Vorbesprechung Serie 9: Aufgabe 2

- Eure Aufgaben:

- Start- und Zielpunkt einlesen
  - Mit Überprüfung auf Gültigkeit

- Rekursive Funktion

`bool sucheWeg(int sx, int sy, int zx, int zy)`  
implementieren

- Ergebnis ist `true` falls ein Weg von  $(sx, sy)$  nach  $(zx, zy)$  existiert, ansonsten `false`
- Mögliche Bewegungsrichtungen:
  - links, rechts, nach oben, nach unten (keine Diagonalbewegungen)

- Vereinfachungen:

- Labyrinthausseiten sind durch Hindernisse begrenzt

# Vorbesprechung Serie 9: Aufgabe 2

## ■ Rekursive Funktion

```
bool sucheWeg(int sx, int sy, int zx, int zy)
```

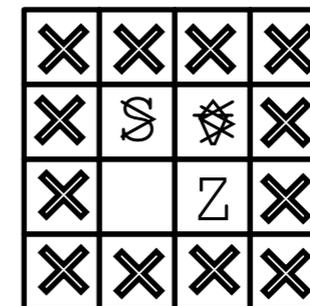
- Ergebnis ist `true` falls ein Weg von  $(sx, sy)$  nach  $(zx, zy)$  existiert, ansonsten `false`

### ■ Abbruchbedingung:

```
 ■ if (sx == zx && sy == zy) // target reached
 return true;
 else if (zeichenAnPosition(sx, sy) != ' ') // invalid Position
 return false;
```

### ■ Alle 4 Möglichkeiten rekursiv nacheinander testen

- Führt ein Schritt nach rechts zum Ziel?
- Führt ein Schritt nach links zum Ziel?
- Führt ein Schritt nach oben zum Ziel?
- Führt ein Schritt nach unten zum Ziel?



- Falls alle 4 Möglichkeiten erfolglos blieben, dann kann das Ziel vom aktuellen Punkt aus nicht erreicht werden

- Markiere den aktuellen Punkt mit `\.'` und gebe `false` zurück