Praktische Aspekte der Informatik

Thomas Löwe
Prof. Marcus Magnor
Introduction

What you need to know.
PADI = Praktische Aspekte Der Informatik

You will learn...
  ... how to program in C++.
  ... how to work with libraries.
  ... how to debug your code.
  ... how to optimize your code.
  ... how to organize your code.
  ... much more!
Why?

• If you’re a bachelor’s student…
  ... you already know Java!
  ... you will have to do the SEP and the Teamprojekt.
  ... you will have to write a Bachelor’s thesis.

• If you’re a master’s student…
  ... you may want to write a Projektarbeit.
  ... you will have to write a Master’s thesis.

• Eventually, all of you will work in the real world!
How?

• **First Part:** Weekly Assignments
  - Brief talk (15-20 min).
  - Work on assignment in small groups.
  - Develop a proposal for a project.

• **Second Part:** Your Project
  - Brief talk (15-20 min).
  - Work on your own project.
  - Give regular updates on your progress.

• **Last week:** Present your Project
  - 15.07. + 17.07.2015
• Your software project…
  … can be anything you want (more or less)
  … must be written in C++!
  … should highlight the skills you have learned.
  … must use at least one external library.
  … must have a visual component.

• Your project will be graded on…
  … function & quality.
  … polish & presentation.
  … whether you’re a Ba/Ma student.

• You alone are responsible for your project!
TU Abalone

- Complete ruleset of the Abalone game.
- Various game setups.
- Local multiplayer.
- Responsive user interface.
- Fancy animations!
- Save / Load / Undo / …
Example: Master’s Project

Closer

- Jump & Run.
- Simple 2D graphics.
- Fancy design and animations.
- Levels loaded from custom format.
- Engine architecture from scratch.

- Various helpful debug views.
- Music and Sound Effects.
Final remarks

• Be present!
  ▪ Otherwise, you might miss important announcements.
  ▪ Most importantly, I cannot help you if you’re not here.

• A computer scientist must work in any environment!
  ▪ You may do weekly assignments on your own computer.
  ▪ Your project should run in the CIP-Pool
    If it does not, then you must have a plausible excuse.
  ▪ Challenge: Set up your project for multiple platforms.

• PADI is a lot of work! Pick a project you love!
C++ Basics

Compiler, Classes, Pointers, Inheritance and more!
Warning!

The following slides are meant to give you a very superficial introduction to C++ basics.

If you want to learn more, have a look at:

http://www.cplusplus.com
http://www.cppreference.com
http://www.learncpp.com
• What is C++?
  ▪ Preprocessor, Compiler, and Linker
  ▪ .h and .cpp files

• First steps
  ▪ “Hello World!” (the boring way)
  ▪ Your first C++ class
  ▪ Pointers & References
  ▪ Inheritance in C++
  ▪ Reading and writing files

• Assignment: “Hello World!” (the CG way)
What is C++?

• Benefits:
  ▪ Object oriented
  ▪ Very similar to Java (which you already know!)
  ▪ Fast, powerful, and widely used
  ▪ Many libraries and much code already available
  ▪ A lot of tutorials and help.

• Drawbacks:
  ▪ Not very intuitive / fast to program
  ▪ No Garbage Collector
  ▪ Error prone
• When creating an executable, your code is...
  ... **preprocessed** ("glued" together)
  ... **compiled** (translated to be machine-readable)
  ... and **linked** (all the parts are connected)

• Different errors can occur at different stages
  ▪ Not always easy to understand.
  ▪ More on that next week.
main.cpp

// Import I/O functionality
#include <iostream>

// This is the main entry point.
int main() {

    // Print "Hello World!" and end the program
    std::cout << "Hello World!" << std::endl;
    return 0;

}
Your first class (.h)

MyClass.h

#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {

public:
    MyClass(); // Constructor
    void doMagic(); // Some member function

private:
    int counter; // Some member variable

};

#endif
#include <iostream>
#include "MyClass.h"

// Namespace::functionName() { definition; }

MyClass::MyClass() { // Constructor
    this->counter = 0;
}

void MyClass::doMagic() { // Some member function
    std::cout << "MyClass::doMagic has been called"
               << ++this->counter << " times."
               << std::endl;
}
main.cpp

#include "MyClass.h"

int main() {

    // Create an object of class MyClass
    MyClass my_object;
    for (int i = 0; i < 10; ++i)
    {
        my_object.doMagic(); // Do magic!
    }

    return 0;
}

Building your code

1. Preprocessor

2. Compiler

3. Linker

Hello
They work just like classes*!

*but they are commonly used for storing data.
#ifndef VECTOR3D_H
#define VECTOR3D_H

struct Vector3D {
    Vector3D(float x = 0, float y = 0, float z = 0) {
        this->x = x;  this->y = y;  this->z = z;
    }
    float x,y,z;
};

#endif
#ifndef VECTOR3D_H
#define VECTOR3D_H

struct Vector3D {
    Vector3D(float x = 0, float y = 0, float z = 0) {
        this->x = x; this->y = y; this->z = z;
    }
    float x,y,z;
};

Vector3D operator+(Vector3D a, Vector3D b) { // a+b
    return Vector3D(a.x+b.x, a.y+b.y, a.z+b.z);
}

#endif
STL Vector

```cpp
#include <vector>

// Create a vector and add some values
std::vector<float> values;
values.push_back(1.1);
values.push_back(2.3);
values.push_back(4.2);

std::cout << "Size of vector: "
  << values.size() << std::endl; // Size: 3
std::cout << "Value at index 1: "
  << values[1] << std::endl; // values[1]: 2.3
values.clear();
std::cout << "Size of vector: "
  << values.size() << std::endl; // Size: 0
```
STL Vector

```cpp
#include <vector>
#include "Vector3D.h"

// A vector of Vector3D objects
std::vector<Vector3D> vectors;

Vector3D v(1.1, 2.3, 4.2);
vectors.push_back(v);

vectors.push_back(Vector3D(1.1, 2.3, 4.2));

There are also lists, sets, maps, and more...
```
Pointers and References

- **Easy to spot:** & and *
  - Reference: `float & value;`
  - Pointer: `float * value;`

- **References are safer to use.**
  - There is no special syntax.
  - You cannot change where they point.

- **Pointers give you more freedom.**
  - You can change pointers at runtime.
  - You have to be careful where you point.
  - You have to clean up after yourself: **new** and **delete**
float fooCopy(float float_copy) {
    return ++float_copy;
}
float fooRef(float & float_reference) {
    return ++float_reference;
}
float fooConstRef(float const & float_const_reference) {
    return float_const_reference + 1;
}
float fooPtr(float * float_pointer) {
    return ++(*float_pointer);
}
float fooConstPtr(float const * float_const_pointer) {
    return (*float_const_pointer) + 1;
}
**const** usually applies to what’s *left* of it...

```cpp
float const value = .5f;        // const float  
float const* ptr;              // pointer to a const float
float *const ptr;              // const pointer to a float
float const*const ptr;         // const pointer to const float
```

... except when it doesn’t...

```cpp
const float value = .5f;         // float const value  
const float * ptr;               // float const* ptr;
const float *const ptr;          // float const*const ptr;
```
• By default, objects are allocated on the **stack**.
  ▪ Automatically deleted at the end of their scope.
  ▪ You can access memory addresses using pointers*
    *but be careful with that!

• If you use **new**, objects are stored in the **heap**.
  ▪ Manually clean up using **delete**.
  ▪ Pointers themselves are often on the stack, i.e. *they are automatically cleaned up*, but the allocated object they point to is not!
  ▪ If no **valid pointer** to an allocated object remains, you have created a **memory leak**!
Explicitly allocate/deallocate memory:

```cpp
Vector3D* my_ptr = new Vector3D(1.1, 2.3, 4.2);
// … do stuff …
delete my_ptr;
```

You can also allocate arrays:

```cpp
// allocate 3 consecutive floats in memory
float* my_ptr = new float[3];
my_ptr[0] = 4.2f; // initialize your values…
[…]
float val = my_ptr[1]; // access the second float
float err = my_ptr[5]; // this might/should crash!
delete[] my_ptr; // deallocated the entire array
```
Important Operators when working with pointers:

- Dereference using . and ->
- Unfortunately, there’s also & and *.

```cpp
Vector3D* vec = new Vector3D;
vec->x = 4.2f;
(*vec).x = 4.2f;

[...]
Vector3D other_vector;
vec = &other_vector;
[...]
float value = fooPtr(&vec->x);
```
Why use pointers?

- You can decide when your variables die.
- You can control memory allocation.
- Pointers can be shared—but beware!
- And there are “smart pointers”.

Helpful advice:
- When passing anything bigger than a primitive, pass a `const&`
- When an argument variable is altered by a function, pass a `pointer`
Inheritance

• Classes may inherit from **multiple** other classes (polymorphism)

• Classes can be abstract (just like in JAVA)

• You can define your own...
  ... Constructors,
  ... Destructors,
  ... and (assignment) operators

• Virtual classes allow method overriding
main.cpp

```cpp
class A() {  // Parent class
    virtual int foo() { return 1; }
};

class B() : public A {  // Child class
    int foo() { return 2; }
};

int main() {
    std::vector<A*> vec;
    vec.push_back(new A);
    vec.push_back(new B);  // an A* pointer is also valid for B*
    for (int i = 0; i < vec.size(); ++i)
        std::cout << vec[i].foo();
    return 0;
}
```
void MyClass::readFile(std::string const& filename) {
    // Open the file for reading
    std::ifstream fin(filename.c_str());
    // Read the title (string)
    fin >> this->title;
    // Read the points (float)
    int num_points;
    fin >> num_points;
    for (int i = 0; i < num_points; ++i) {
        float value;
        fin >> value;
        this->values.push_back(value);
    }
    fin.close();
}
```cpp
void MyClass::writeFile(std::string const & fname) const {
    // Open the file for writing
    std::ofstream fout(fname.c_str(), std::ios::out);

    // Write the title and number of points
    fout << this->title + "*1.5" << std::endl;
    fout << this->values.size() << std::endl;

    // Write the point values (multiplied by 1.5)
    for (int i = 0; i < values.size(); ++i)
        fout << (values[i]*1.5f) << std::endl;

    fout.close();
}
```
Wavefront .obj file format

# List of vertices (nodes), with (x,y,z) coordinates
v 0.25 0.53 0.763  # vertex 1
v ...  # vertex 2
v ...

# List of polygons (faces), with vertex indices
f 1 2 3  # triangle
f 2 3 4
f 2 5 6 4  # quad
f ...
world.txt

block 0 -0.125 0 20 0.25 20 # x,y,z, width, height, depth
sphere 8 6 7 2 16 # x,y,z, radius, tessellation
sphere 8 8 7 1.5 16
block 8 3 7 0.5 6 0.5
block -6 2.5 2 6 5 8
block -4 3.5 -5 10 7 6
Assignment

A. Look at the examples
   Have a look at, compile, and understand all examples.

B. Hello World!
   Implement a small C++ program that reads the world.txt file
   and converts it to an .obj file.

   Look at assignment_stub/main.cpp for more information.

   Hint: If you’re stuck, the internet will help!

C. Have Fun!
   How about adding more object types? (Cylinders, Pyramids, …)
   How about adding rotation?
   …
Closing thoughts

• PADI Philosophy
  ▪ Gap knowledge.
  ▪ Practical skills that glue together different aspects of CS.

• Heterogeneous Audience
  ▪ Some of you know nothing, some know a lot.
  ▪ Adjust your ambitions for the final project.
  ▪ Help each other!

• Improve your individual programming skills!
Wichtig!

Die Termine für die abschließenden Kolloquien sind:

15.07.2015  15:00 Uhr
17.07.2015  13:15 Uhr

Ihr müsst PADI beim Prüfungsamt anmelden!