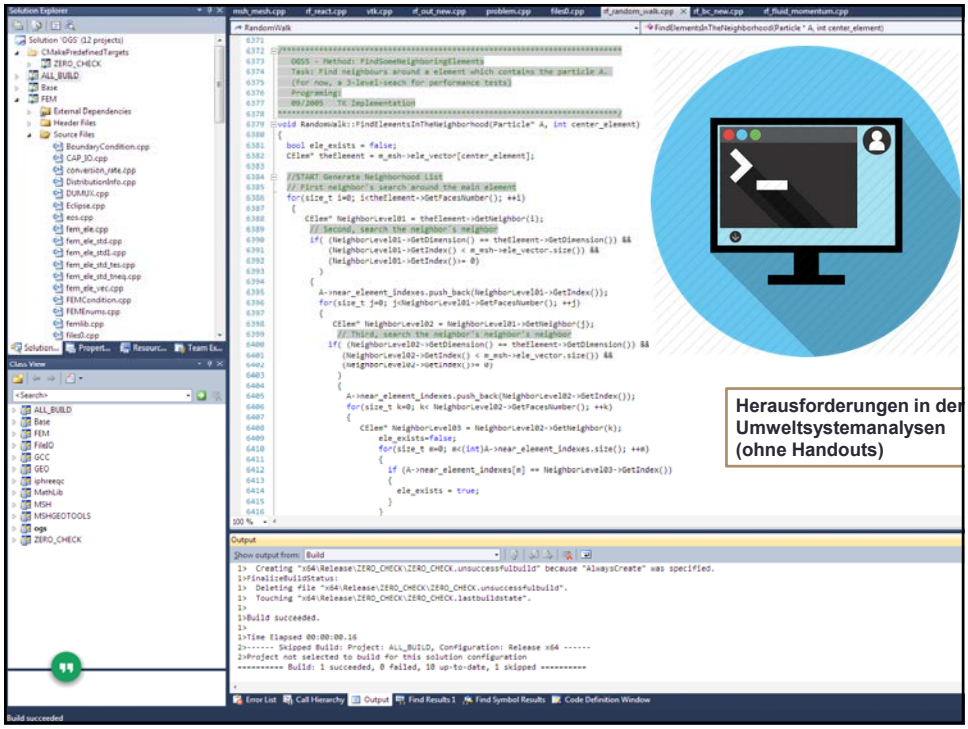
HYDROINFORMATIK 1

C++ Programmieren in Natur- und Ingenieurwissenschaften

(Einführung)

Dr. Thomas Kalbacher
Department für Umweltinformatik
Helmholtz-Zentrum für Umweltforschung GmbH - UFZ
thomas.kalbacher@ufz.de / http://www.ufz.de



- Rückblick bzw. Quiz
- Object-Oriented Programming
  - Grundlagen, Konzept, Klassen

I hear, and I forget;
I see, and I remember;
I do, and I understand.

Confuzius, 551-479 b.C.

Hydroinformatik I

C++ Einführung

Kurzer Blick zurück

Some Basics, Pointer & Operators

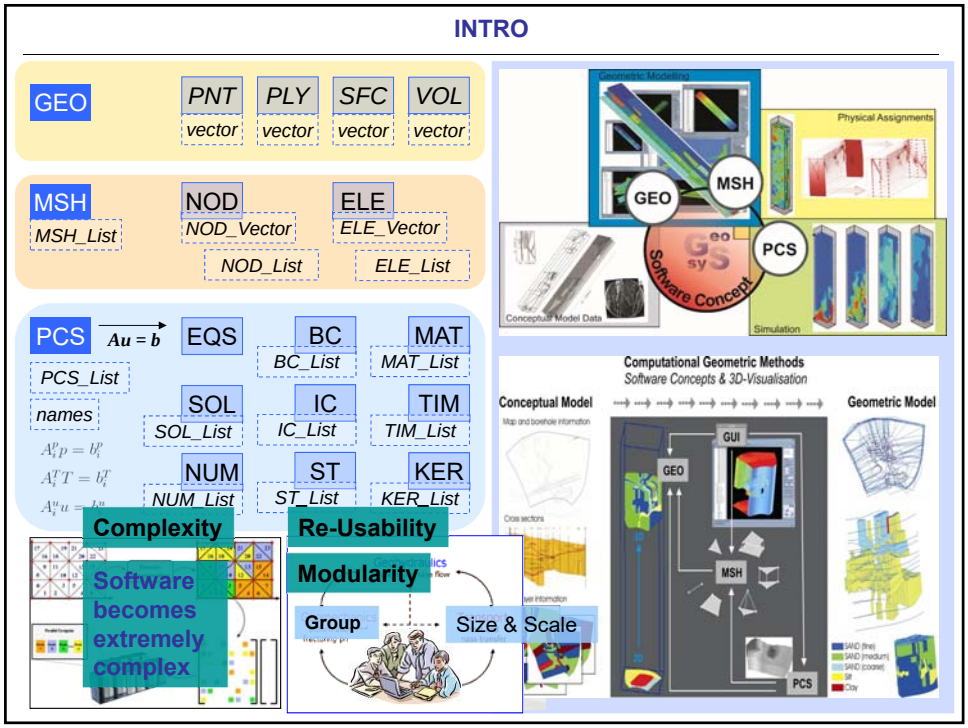
Quiz: What is the output?

```
int p(char * str)
{
  printf("%s\n", str);
  return 0;
}
int global = p("global");
void f()
{
  p("f");
  static int local = p("local");
}
int main()
{
  p("start"); f(); f(); f(); p("stop");
  return 0;
}
```

Hydroinformatik I

C++ Einführung

Object-Oriented Programing



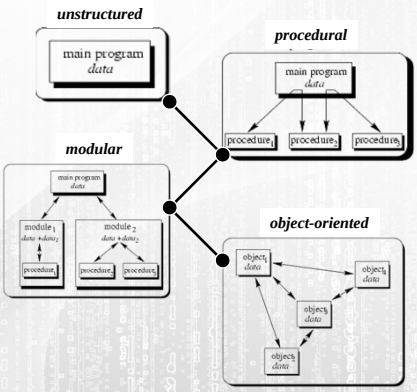
Introduction

What are Software / Programming Concepts?

INTRO CONCEPTS OOP GEO MSH PCS

Names / Terms

- *Unstructured Programming,*
- *Procedural Programming,*
- *Modular Programming and*
- *Object-Oriented Programming.*



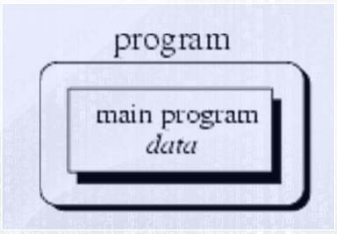
Concepts: Unstructured Programming

Unstructured Programming:

usually the way how beginner learn to make "small" programs.

*The **main** program*

- *stands for a sequence of commands*
- *modifies data directly*
- *operates on global data.*



Disadvantage:

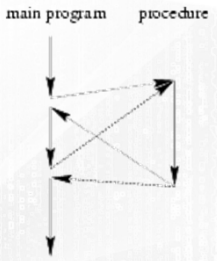
large programs + same sequences are needed at different locations
→ *sequences must be copied*

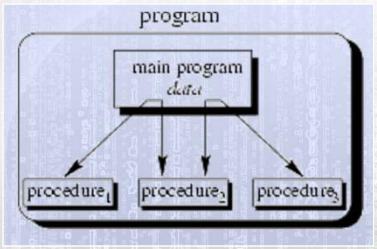
→ Idea: extract these sequences, name them and offering a technique to call and return from these *procedures*.

Concepts: Procedural Programming

Procedural Programming :

- *A procedure call is used to start the procedure.*
- *After the sequence is processed, flow of control proceeds right after the position where the call was made.*
- *The main program is responsible to pass data to the individual calls.*
- *Now a program can be viewed as a sequence of procedure calls.*
- *The flow of data can be illustrated as a hierarchical graph, a tree, for a program with no subprocedures*





Concepts: Procedural Programming

INTRO
CONCEPTS
OOP
GEO
MSH
PCS

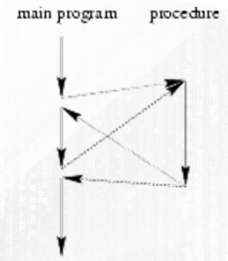
Procedural Programming :

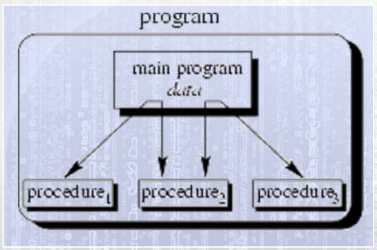
Advantages:

- *If a procedure is correct, every time it is used it produces correct results.*
- *In cases of errors you can narrow your search to those places which are not proven to be correct. After the sequence is processed, flow of control proceeds right after the position where the call was made.*

Disadvantages:

- *No usage of subprocedures, general procedures or groups of procedures.*





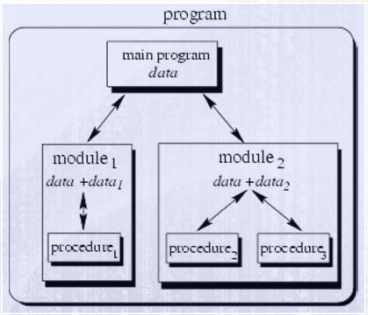
Concepts: Modular Programming

INTRO
CONCEPTS
OOP
GEO
MSH
PCS

Modular Programming :

With modular programming

- Procedures of a common functionality are grouped together into separate modules.
- A program now divided into several smaller parts which interact through procedure calls and which form the whole program.
- Each module can have its own data (1 internal state).
- Modules manage their internal state by calls to their procedures.



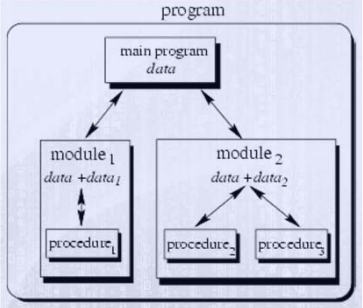
Concepts: Modular Programming

INTRO CONCEPTS OOP GEO MSH PCS

Modular Programming :

Disadvantages:

- Explicit Creation and Destruction: *more complex data structures (vectors, lists, arrays, etc) must be explicit created and destructed*.
- Decoupled Data and Operations: *Decoupling of data and operations leads usually to a structure based on the operations rather than the data.*



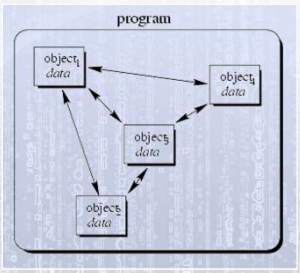
Concepts: Object-Oriented Programming

INTRO CONCEPTS OOP GEO MSH PCS

Definition:

Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.

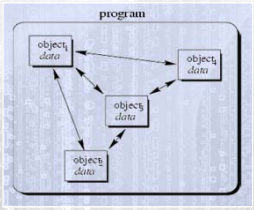
(From Wikipedia, the free encyclopedia)



Concepts: Object-Oriented Programming

Structure and Relationships:

- Definition of data types of data structures, but also the types of operations (methods, functions) that can be applied to the data structure.
- The data structure becomes an object that includes both data and functions.
- Relationships between one object and another: For example, objects can inherit characteristics from other objects.



OOP: Basic Motivation

The basic motivation for an object-oriented structure is

- **Re-Usability:**
  - Existing objects (perhaps written by another software developer), can be simply used.
- **To control Complexity:**
  - Hardware and software became increasingly complex.
  - Long term developments make software more complex.
  - Researcher must maintain the software quality.
- **Modularity:**
  - The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

Real-world objects share two characteristics:

1. *They all have **state**:*
   (e.g. a dog: name, color, breed, hungry)
2. *They all have **behavior**:*
   (e.g. a dog: barking, fetching, wagging tail)

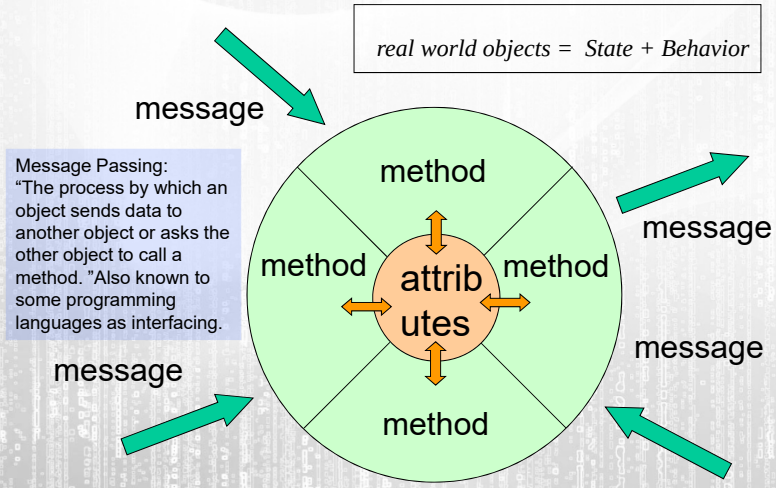
Bicycle **state**: current gear, current pedal cadence, current speed

Bicycle **behavior**: changing gear, changing pedal cadence, applying brakes

OOP: Objects

Object = Attributes + Methods

real world objects = State + Behavior

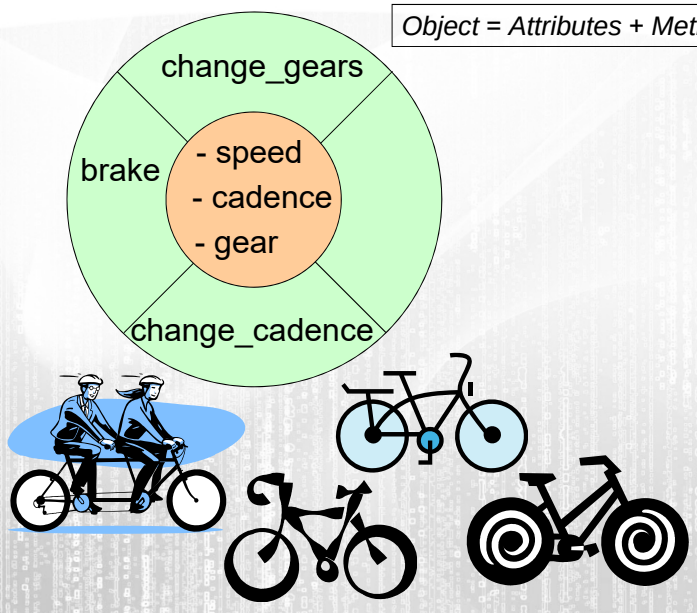


Message Passing:
"The process by which an object sends data to another object or asks the other object to call a method. "Also known to some programming languages as interfacing.

INTRO CONCEPTS OOP GEO MSH PCS

OOP: Objects

Object = Attributes + Methods



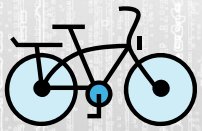
INTRO CONCEPTS OOP GEO MSH PCS

OOP: Objects

INTRO CONCEPTS OOP GEO MSH PCS

- Objects vary in complexity.
- Some objects can contain other objects.
- The object remains in control of how the outside world is allowed to use it by:
  - attributing the state (current speed, current pedal cadence, and current gear)
  - and providing methods for changing that state.

```
//example bicycle only has 6 gears
int ChangeGear(int new_gear)
{
     if (new_gear > 6)
     return 6;
     if (new_gear < 1)
     return 1;
     else
     return new_gear;
}
```

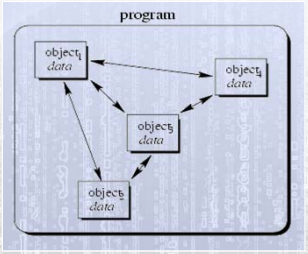


OOP: Objects

INTRO CONCEPTS OOP GEO MSH PCS

Object = Attributes + Methods

- *Attributes* : data that describe the internal status of an object
  - "*member variables*" in C++
  - inaccessible from outside → *Encapsulation*
- *Methods* : functions which can access the internal status of an object
  - "*member functions*" in C++
  - accessible from outside
  - manipulates attributes



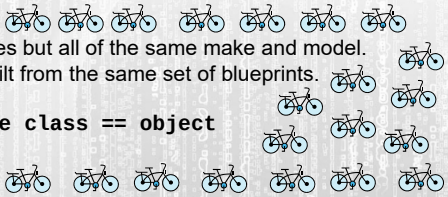
OOP: Class

INTRO CONCEPTS OOP GEO MSH PCS

What is a class?

- A class defines the abstract characteristics of an object, including the object's attributes and the object's methods.
- The characteristics of the class should make sense in context.
- The code for a class should be relatively self-contained (generally using encapsulation).
- Collectively, the properties and methods defined by a class are called members.

Example:  Thousands of bicycles but all of the same make and model. Each bicycle was built from the same set of blueprints.



Each bicycle == instance of the class == object
Blueprint == class.

OOP: Class

In C++:

INTRO CONCEPTS OOP GEO MSH PCS

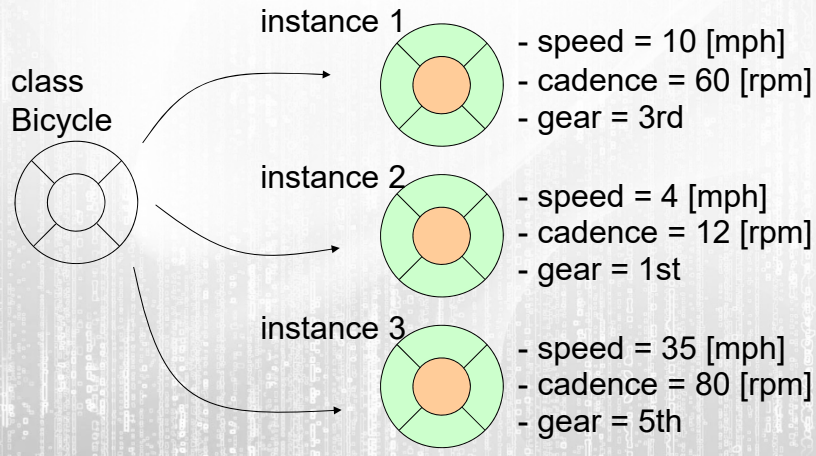
```
class Bicycle {
private:
    float speed;
    float cadence;
    int gear;
public:
    void change_gears(int gear);
    void break();
    void change_cadence(float cadence);
};
```

• A class is a template definition of the methods and variables.

• An object is a specific instance of a class; it contains real values instead of variables. The object or class instance is what you run in the computer.

OOP: Class

Different instances can have different values of member variables

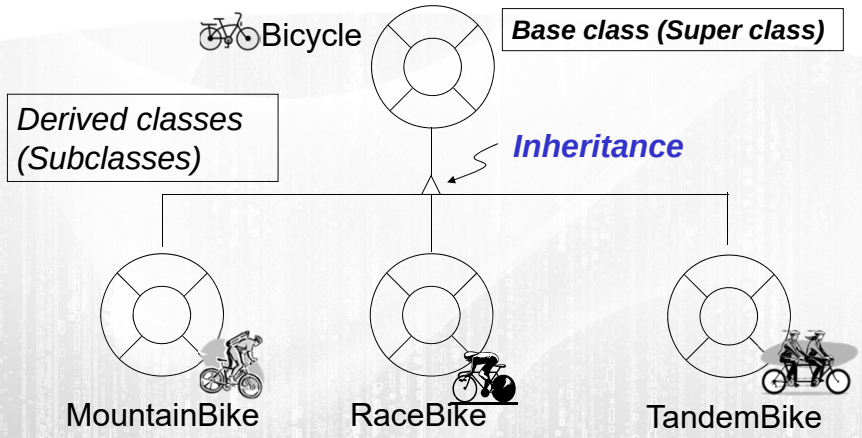


OOP: Inheritance

A class can have **subclasses** that can "**inherit**" all or some of the characteristics of the class.

- In relation to each subclass, the class becomes the **superclass**.
- Subclasses can also define their own methods and variables that are not part of their superclass.
- The structure of a class and its subclasses is called the **class hierarchy**

OOP: Inheritance



'Subclasses' are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.

OOP: Inheritance

Advantages:

- Selective reuse of classes, by creating new derived classes with new included features.
- Create new classes specific to new problem domains.

- Increases the Extensibility of programs.
- Class hierarchy allows to construct frameworks.
- Simplifies the program structure and enables generic, "logical" programming.

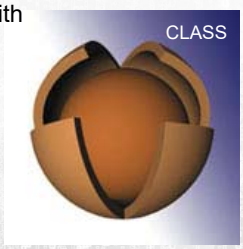
OOP: Encapsulation

INTRO CONCEPTS OOP GEO MSH PCS

Encapsulation:

Objects reveal their "outside" through “touchable” behaviour, but keep their "inside" hidden.

Methods form the object's interface with the outside world;

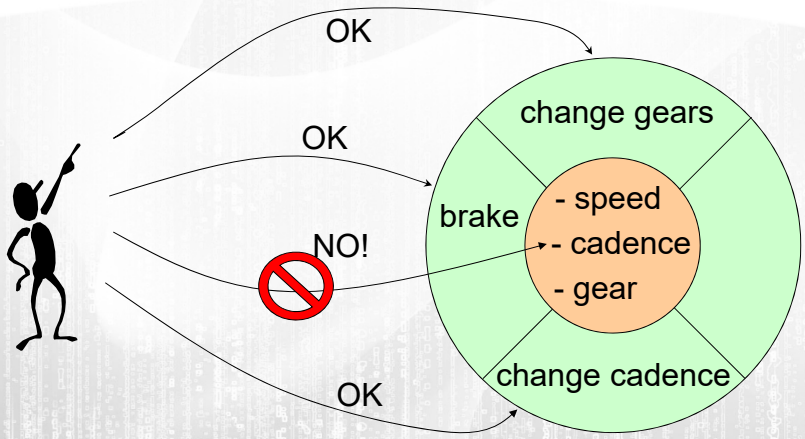


The class keeps the functional details hidden from objects that send messages to it.

We do not need to know how an object works to know how it behaves.

OOP: Encapsulation

INTRO CONCEPTS OOP GEO MSH PCS



- **Objects should not access to attributes of other objects directly.**
- **Access to the attributes is done by methods!**
- **Such methods ensure the consistency of the attributes.**

OOP: Encapsulation

INTRO CONCEPTS OOP GEO MSH PCS

With Encapsulation, we

- **separate what's relevant from what's not.**
- **collect what's relevant in one location.**

Why build forts, castles, bunkers, customs, etc?
To ensure that what's outside doesn't mess with what's inside.

▪**Managing Change**

Why put up roadblocks or prisons?
To ensure that what's inside stays there.

▪**Protecting Data**

Why put all the fuses in one fuse-box?
To localized the problem when it occurs.

▪**Managing Complexity**

OOP: Abstraction

INTRO CONCEPTS OOP GEO MSH PCS

In Computer Science, Abstraction is a mechanism to reduce details so that one can focus on a few concepts at a time.

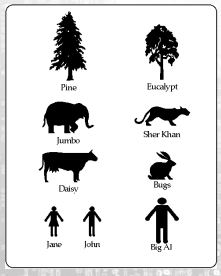
Abstraction is

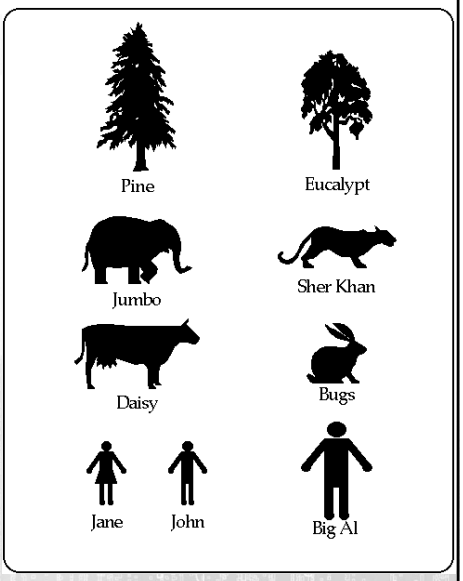
- simplifying complex reality by modelling classes appropriate to the problem.
- working at the most appropriate level of inheritance.
- An Art of concentrating the essential and ignoring the non-essential.

OOP: Abstraction

INTRO CONCEPTS OOP GEO MSH PCS

Lets "chunk" things by identifying significant commonalities (whilst ignoring trivial differences)...

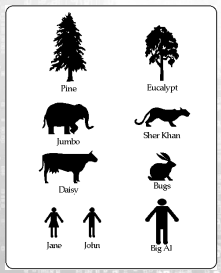


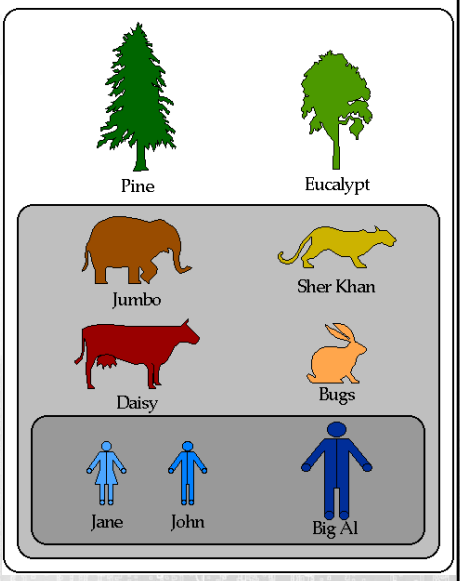


OOP: Abstraction

INTRO CONCEPTS OOP GEO MSH PCS

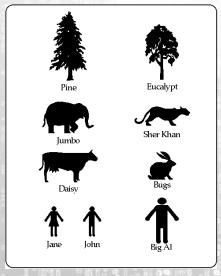
How we choose to abstract depends on what we consider to be "significant"... plants, animals, dangerous animals

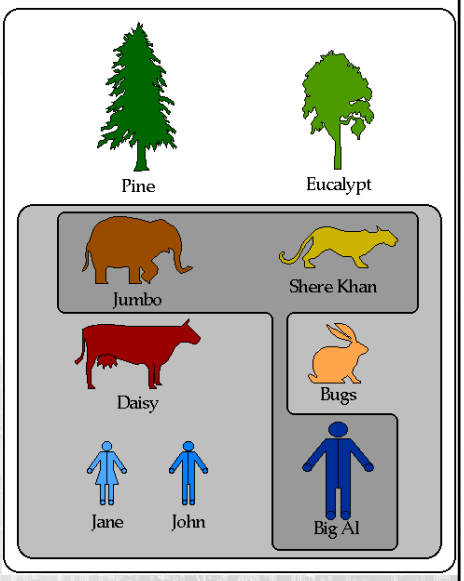




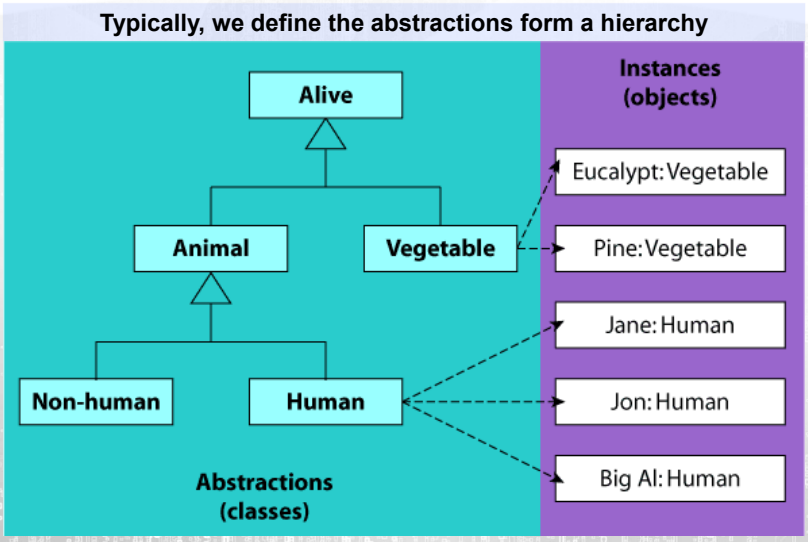
OOP: Abstraction

This leads to classification:
"All men are animals" by which we mean: "All men share certain attributes with all other entities which we also label 'animals'"





OOP: Abstraction

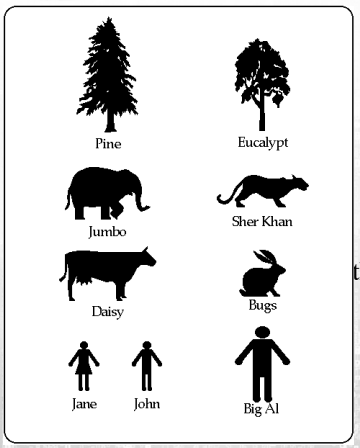


OOP: Abstraction

Types of Abstraction:

Abstraction of:

- Appearance (e.g. size)
- Structure (e.g. 2 eyes)
- Functionality (e.g. can carry things)
- Privilege (needs protection)
- Purpose (high population)



OOP: Polymorphism

Polymorphism :

Objects belonging to different data types can respond to calls of methods of the same name.

A function that can be applied to values of
- different types of arguments and/or
- different number of arguments

is known as a *Polymorphic Function* or *Overloaded Function*.

```
// Example abs with different argument types
int abs(int n);
long abs(long n);
double abs(double n);
// Example with different numbers of arguments
int strcpy(char *str1, char *str2, short unsigned n=65535);
// second overloaded function
int strcpy(char *str1, char *str2);
```

Compilation: the technical way

A Project is build by following pseudo code:

for all .cpp-files x.cpp *in* Project:
    preprocess x.cpp
    compile x.cpp :
        creates file x.o

link all x.o and some libraries:
    creates file x.exe

translation unit

The Preprocessor

- Calls for the Preprocessor start with #.
- #include connects a Header-File.

```
#include <stdio.h>
```

- #define defines a Macro: The Keyword will be replaced in the whole program by the given "text".

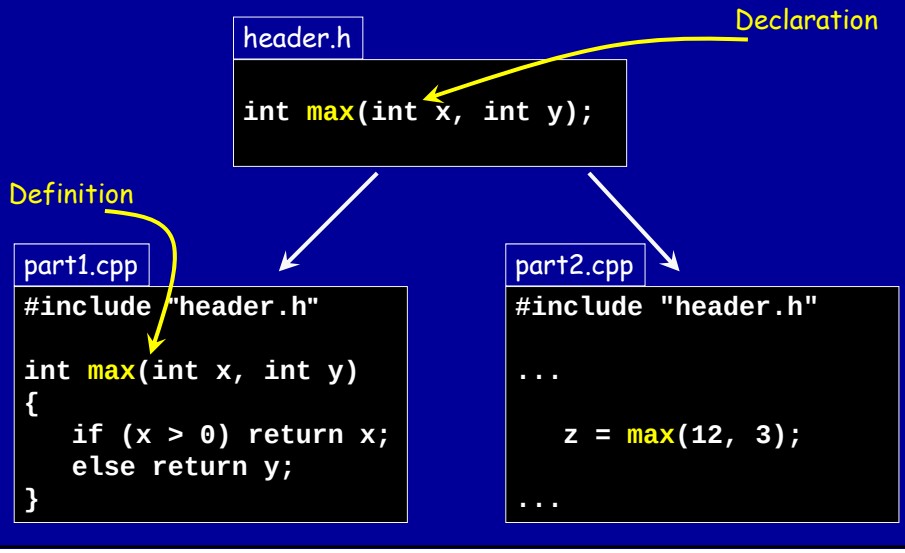
```
#define PI 3.141596
```

More than One File

A C/C++-Project can consist of more than one file:

- .c-file: source code file.
  (in C++: also .c or .cpp)
- .h-file: header file.
  (in C++: also .hpp)

Distribution of Source Code



Declaration & Definition

Declaration of X =

Description of "what is X"

Definition of X =

Declaration of X
+ Creation of X

» X can be declared many times, but it can be only defined once. «
Header files should contain only declarations but no definitions because all linked source code files would define a new X.

Declaration & Definition (2)

Declaration **Definition**

- Variables:

```
extern int x;
```

```
int x;
```

- Functions:

```
int add(int, int);
```

```
int add(int a, int b)
{
   return a + b;
}
```

Validation of Declarations

Rule 1:

»A symbol in the source code is only known below its declaration.«

Rule 2:

»A Declaration, which is placed inside a { } -Block, is only locally valid within this block.«

One Definition Rule

Rule: » Each Entity (Variable, Type, Function, Class) can be only defined once. « (One Definition per Translation Unit)

Remarks:

- Multi definitions are possible but can produce trouble for the linker.
- Local Variables (in different functions) can have the same name, of course.
- C++ Classes: Here we can redefine classes in each translation unit because the definition of the class do produce a entity which is used by the linker.

Member-Functions

Beside the Member-Variables C++ knows additionally Member-Functiones:


```
class Hamster
{
   int age;
   char name[256];
   void feed();
};

Hamster billy;
billy.feed();
```

Declaration and Definition

Declaration **Definition**

- Classes:

```
class Hamster;
```

```
class Hamster
{
   int weight;
   int feed(int);
};
```

- Member-Functions:

```
class Hamster
{
   int feed(int);
};
```

```
int Hamster::feed(int x)
{
 weight += x;
}
```

Programming Classes

Typical Procedure:

- Inside the .h-file: Class Definition
- Inside .cpp-file: Definition of the Member-functions.

Including the header file to many cpp files has no influence to the memory, because class definitions do not allocate memory and produce no code. Anyway, a class can not be included twice in a cpp file.

Some Remarks . . .

- Multiple Class Definitions disturb the compiler but not the linker, because there is no code inside the definition which must be linked.
- It is very useful to define classes in header files und to include these header files to the cpp files!
- Of course, member-functions can be defined as well inside the class definition and are therefore so called **inline**.
- External defined member-functions can't be multiple defined. Multiple member function definition would produce an error inside the linker – beside the member function is defined inline.

public and privat

private protects the members!

No Access from Outside allowed:


```
class Hamster
{
private:
    bool is_happy;
public:
    void feed() { is_happy = true; }
};

Hamster billy;
billy.is_happy = true; //ERROR!
```

Constructors

Constructors = Special Member-Function, that is automatically called with the activation of the object:

```
class CHamster
{
   public:
    CHamster();// constructor
    ~CHamster(); // destructor
   int age;
};
```

Constructors don't return any types (not even **void**). They are used to initialize the object.

Destructors

Destructors = Destructors are called during the Object will be destroyed.

```
class CHamster
{
   public:
    CHamster();// constructor
    ~CHamster(); // destructor
   int age;
};
```

Definition of Constructors /Destructor

C++

```
CHamster::CHamster()
{
}
CHamster::~CHamster()
{
}
```

Example

Constructors and Destructors can use Arguments


```
class Hamster
{
    Hamster (char * name = "Billy") {
          Name = name;
          printf("Hamster is born! :-)\n");
    }
    ~Hamster () {
          printf("Hamster dies! :-(\n");
    }
    char * Name;
};
```

Example


```
void hamsterlife()
{
    Hamster billy;
    printf("Welcome the new Hamster!\n");
    printf("It's name is %s.\n", billy.Name);
}
```

Start: `hamsterlife()`

Output:

```
Hamster is born! :-)
Welcome the new Hamster!
It's name is Billy.
Hamster dies! :-(
```

Member-Functions Definition

Member-Functions need a definition.

1. Possibility: Inside the Class definition

```
class Hamster
{
   void feed()
   {
       //now the hamster gets some food!
   }
};
```

Member-Functions Definition

2. Possibilty: Outside the Class definition


```
class Hamster
{
   void feed();
};

void Hamster::feed()
{
   //now the hamster gets some food!!
}
```

Declaration and Definition

Declaration **Definition**

- Classes:

```
class Hamster;
```

```
class Hamster
{
    int weight;
    int feed(int);
};
```

- Member-Functions:

```
class Hamster
{
   int feed(int);
};
```

```
int Hamster::feed(int x)
{
 weight += x;
}
```

The Member-Function knows the Object

A Member-Function knows to which object it belongs:


```
class Hamster
{
    void feed()
    {
        ++weight;
    }
    int weight;
};

Hamster billy;
billy.feed();
```

⇐ increase **billy.weight**

The this-Pointer

Inside the member-function the **this**-Pointer points on the according Object:

C++

```
class Hamster
{
    void feed()
    {
        ++( this->weight);
    }
    int weight;
};
```

this has the Type **Hamster ***

Overloading of „normal" Functions

Different Functions with same Names:

C++

```
int max(int a, int b)
{
  if (a > b) return a;
  return b;
}

int max(int a, int b, int c)
{
  return max( max(a, b), c );
}
```

Overloading: Rules

The overloaded functions must be different by:

- Number of Arguments

  or

- Type of the Argument at position i.