Classes

Christian Schumacher, chschuma@inf.ethz.ch

Info1 D-MAVT 2013

- Object-Oriented Programming
- Defining and using classes
- Constructors & destructors
- Operators
- friend, this, const

Example

- Student management system
  - Students consist of different types of data (name, age, courses)
  - Can be modeled as struct

```
struct Student
{
    char* name;
    int age;
    int courses[50];
    bool isMaster;
    bool isImmatriculated;
};
```

```
name = "Bob"
age = 21
courses = { 2, 4 }
isMaster = false
isImmatriculated = true
```

```
name = "John"
age = 24
courses = { 7, 10, 12, 33, 71 }
isMaster = true
isImmatriculated = true
```

Example

- Function to change student status has to have student passed as argument

```
struct Student
{
        char* name;
        int age;
        int courses[50];
        bool isMaster;
        bool isImmatriculated;
};
```

```
void exmatriculate(Student &s)
{
    s.isImmatriculated = false;
}
```

- Function is specific to `Student` data type, but defined separately
    - Connection of data and functionality should be reflected in code

→ CLASSES

Object-Oriented Programming

- Abstraction
  - model problems with multiple objects that interact
- Encapsulation & Data Hiding
  - hide complex implementation
- Inheritance
  - design new classes using already existing member variables and functions of already defined classes
- Polymorphism
  - write a function to compare fruits and the program decides which function to call based on whether you compare oranges or apples
- Reusability and code modularity

General Remarks

- Classes exist only in C++, not in C
- Classes (`class`) consists of:
  - Set of *member* variables (like `struct`)
  - Set of *member* functions (so-called *methods*)
  - Defines visibility of *members* (`public` / `private`)
- Main difference to structs:
  - In classes, member access is private, if not otherwise specified
  - In structs, member access is public, if not otherwise specified

Defining Classes

class keyword identifies class definition

Name of the class

private keyword defines class members that can be accessed only by the class

public keyword defines class members that can be accessed by everyone

```
class Complex
{

private:
    double real;
    double imag;

public:
    void set(double r, double i);
};
```

Object Declaration

- After declaring a class, it is available as a new type
- We can use it to define variables
  - The instance of a class is called «object»

```
Complex number1;
Complex number2;
```

- We can also call the public member functions

```
number1.set(1, 0);
number2.set(4.93, -1);
```

- Syntax:

```
object.function_name(argument);
```

Calling member functions

- Objects can also be created with `new`

```
Complex* cPtr = new Complex;
```

- Dereferencing with `*`

```
(*cPtr).set(3.0, 2.0);
```

- Shortcut using `->`

```
cPtr->set(3.0, 2.0);
```

- Syntax: `object_pointer->function_name(argument);`

Implementing Member Functions

- So far we only defined the member functions. To define what they do we need to implement them
- Member functions are basically regular functions, but we have to use the scope operator to define to which class they belong

```
void Complex::set(double r, double i)
{
    real = r;
    imag = i;
}
```

Implementation of the «set» member function of class «Complex»

- Member functions can use all private member of the class they belong to

Header Files

- Usally the class is written in two separate files:
- Header file
  - **complex.h**
  - Definition of members

```
class Complex
{

private:
    double real;
    double imag;

public:
    void set(double r, double i);
};
```

- Body file
  - **complex.cpp**
  - Includes complex.h
  - Implementation of member functions

```
#include "complex.h"

void Complex::set(double r, double i)
{
    real = r;
    imag = i;
}
```

Constructors

- How to initialize objects?
- Structs were initialized as follows:

```
Student stud = { “Hans”, “Heiri”, 13123456 };
```

- For classes this is not possible anymore since private member variables cannot be accessed from outside anymore!
- Solution: The object has to initialize itself when created

Constructors

- Constructors are methods which are automatically called when an object is created.

- The default constructor always exists:

```
Complex c;
Complex d = Complex();
```

- If not redefined, it initializes all member variables with their own default constructor
- Can have multiple constructors for increased flexibility

Constructors

- Default constructor can be redefined
  - The constructor is a special member function with no return type and the same name as the class

```
class Complex
{
private:
    double real;
    double imag;

public:
    Complex();
    void set(double r, double i);
};
```

Complex.h

```
Complex::Complex()
{
    real = 1;
    imag = 0;
}
```

Complex.cpp

Constructors

- Constructors can also take arguments

Complex.h

```
class Complex
{
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double r, double i);
    void set(double r, double i);
};
```

Complex.cpp

```
Complex::Complex(double r, double i)
{
    real = r;
    imag = i;
}
```

- Calling the new constructor:

```
Complex c1 = Complex(1.0, 2.0);
Complex c2(1.0, 2.0);
```

- Every definition of a constructor turns default constructor invalid
  - Redefine the default constructor yourself

Destructors

- Second type of special member functions
- Called automatically when an object is “destroyed”
- Destructors can be used to clean up memory that was allocated by the class and is not used anymore

Complex.h

```
class Complex
{
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double r, double i);
    ~Complex();
    void set(double r, double i);
};
```

Complex.cpp

```
Complex::~Complex()
{
    // destroy the world.
}
```

- Only one destructor for each class
  - Has the same name as the class with a tilde “~” in front.
- Destructors have no return value and no arguments.

Destructors

- Calling destructors
  - Automatically called during `delete`

```
Complex *c = new Complex;
...
delete c;
```

  - Called at end of scope in which object was created

```
{
    Complex c;
    ...
    // --> call to destructor of `c`
}   // --> end of scope of `c`
```

const

- Indicates that no data is modified
- Variables:

```
int j = 5, k = 6;
const int i = 1;
i = 2; // ERROR

int * const i = &j;
i = &k;   //ERROR
*i = k;    //ok
```

- Functions: `const Complex getSum(const Complex a) const;`
  - Return value is const
  - Parameter is const
  - Function is const
    - no members variables are modified
    - can call only other const methods

this & friend

- this
  - Pointer to the current object

```
Complex::Complex(double real, double imag2)
{
    this->real = real;  // ‘this’ necessary
    imag = imag2;       // ‘this’ not necessary
}
```

- Friends
  - Allow access to private functions and variables from other classes

```
class Complex
{
public:
    friend void myFunction(); // friend function
    friend class MyClass;     // friend class
}
```

  - `myFunction` can access private members of `Complex`
  - All member functions of `MyClass` can access private members of `Complex`

Operators

- Interaction between objects
- Could be solved this way:

```
Complex c(8.3, 2.4);
Complex d(0.5, 4.1);

c.add(d);      //Member functions that implement
c.sub(d);      //the four basic arithmetic
c.mult(d);     //operations
c.div(d);
```

- But would be cool to use it this way:

```
Complex e;

e = c + d;
e = c – d;
e = c * d;
e = c / d;
```

Operators

- Operators (+, *, …) can be defined, just like functions
- Use `const` to indicate immutability
  - The operands should not be changed by accident!

```
class Complex
{
private:
    double real;
    double imag;

public:
    void set(double r, double i);
    Complex operator+(const Complex &c2) const;
};

Complex Complex::operator+(const Complex &c2) const
{
    Complex sum;
    sum.real = real + c2.real;
    sum.imag = imag + c2.imag;
    return sum;
}
```

Operators

- Now the addition operator can be used

```
Complex a;
Complex b;

// Call as operator:
Complex s1 = a + b;
```

- Or the operator can be called as a normal function

```
// call as function:
Complex s2 = a.operator+(b);
```