Computational Engineering 2014 - D-MAVT

- What is Computational Engineering?
  - Computational Methods + Programs => Applications
  - Solve numerical problems **accurately** and **efficiently** using **computer programs**
- Today: introduction to “programs”
  - Recap of **C++ basics** taught in “Informatik I” by Prof. Gross
  - Demonstration of UNIX/Linux shell (**Terminal**) and programming examples
- Next week Prof. Koumoutsakos will introduce Computational Engineering and the course in general
- More infos on course webpage:
  http://www.cse-lab.ethz.ch/index.php/teaching/42-teaching/classes/595-computation-engineering-2014
  - Coming Soon: Exercise groups, Exercise 0, Updated VirtualBox from Info I

Exercises and practice hours

- We will work on **Linux** machines provided by ETH. Full support will be provided **exclusively** for such environment.
- The **final exam will be carried out on ETH machines**
  - many available in computer rooms: https://www1.ethz.ch/id/services/list/comp_raum_stud/arbeitsraeume
- **Makefile**-based projects
- **Goal 1: C++** programming for simulations
  - simulations within the realm of particle methods
- **Goal 2: work with external tools**, open-source libraries, shell scripts and makefiles

Exercises and practice hours

- How to pass this course:
  - Pass the exam and that's it...
  - No "Testat", no pass/fail, no exercise corrections => Autonomous work
  - **Programming can only be learnt by doing it**
  - **WISH :** If you have done the exercises **on your own** you will PASS the class
- Exercise sessions
  - Hints for exercises will be provided
  - Programming demonstrations might be given <= **ask questions to your assistant**
- Exercises are not corrected but a practice hour will be offered in the computer room **HG E27** (Thursdays, 17:15 - 18:00)
  - Exercises will be put online Mondays before the exercise sessions
  - Master solutions for the exercises will be put online the following week

C++ Basics

- Informatik I slides available at http://graphics.ethz.ch/teaching/info1_13/notes.php
  - “Fast Forward (May 31)” (1-22), “Functions II (April 18)” (1-8), “Klassen (April 25)”
- Important knowledge for this class:
  - **cout** and other streams => important for test codes (console and file output)
  - Pointers and dynamic memory allocation (**new, delete**)
  - References **&** (vs Pointers *)
  - **classes**: constructors, initialization lists, destructors, operators
  - **function calls**: pass parameters and results by value & by reference
  - **const** keyword
  - **Abstraction** (no need to look at and understand every line of code...ever!)
- Later in this class: STL, inheritance and templates
- Trained mainly by doing exercises

const revisited

```
int i = 5;
const int ci = 10;
ci = i; // NOT OK (ci is constant)
i = ci; // OK (can copy constant value)
const int ci2 = i; // OK

int * pi = &i;
*pi = 5;  // OK
pi = &i;  // OK
const int * pi2 = &i;
*pi2 = 5; // NOT OK (*pi2 is constant)
pi2 = &i; // OK (pi2 is not constant)
int const * pi3 = &i; // same as pi2
int * const pi4 = &i;
*pi4 = 5; // OK (*pi4 is not constant)
pi4 = &i; // NOT OK (pi4 is constant)
const int * const pi5 = &i;
*pi5 = 5; // NOT OK
pi5 = &i; // NOT OK
```

- const = assigned once, never changed anymore
  - assignment itself can be done using a runtime variable!
- for complex pointer types **const refers to whatever is left** (!) of the keyword (unless const is left-most: then it refers to first part)

Classes - what's this?

```
class Stock {
  float value;
public:
  float getValue() {
    return value;
    // equivalent: return this->value;
  }
  void setValue(float f) {
    value = f;
    // equivalent: this->value = f;
  }
  Stock* getMax(Stock* other) {
    if (other->value > value) {
      return other;
    } else {
      return this;
    }
  }
};
```

```
int main() {
  Stock s1;
  s1.setValue(10);
  Stock s2;
  s2.setValue(20);
  Stock* smax1 = s1.getMax(&s2);
  Stock* smax2 = s2.getMax(&s1);
  cout << smax1->getValue() << endl;
  cout << smax2->getValue() << endl;
  return 0;
}
```

Side note: can be done more elegantly using references "Stock &"

- object x of type X => "X x;" => call with "x.f()"
  - pointer: "X * x;" => call with "x->f()" (shorthand for "(*x).f()")
- whenever "x.f()" is called function f gets hidden argument "this" with *this = x

Classes - constructors and initialization lists

simple class

```
class Foo {
  int size;
};
```

by default: private attributes

class with constructor

```
class Foo2 {
  int size;
public:
  Foo2(int N) { size = N; }
};
```

class with constructor and init. list

```
class Foo3 {
  int size;
public:
  Foo3(int N): size(N) { }
};
```

- If no constructor implemented, C++ provides one => “Foo f;” OK
  - If any constructor implemented, those have to be used => e.g. “Foo2 f2(2);”
- Initialization list
  - consider difference between “int i; i = 1;” and “int i = 1;” (or equivalently “int i(1);”)
  - same difference when using initialization list
  - more relevant for custom types like Foo2:
    - can only work with initialization list since “Foo2 f;” is undefined but “Foo2 f = N;” (or equivalently “Foo2 f(N);”) is OK

```
class FooFoo {
  Foo2 f;
public:
  FooFoo(int N): f(N) { }
};
```

Classes - destructor

bad class

```
class FooB {
  int size;
  float * data;
public:
  FooB(int N): size(N) {
    data = new float[N];
  }
};
```

good class

```
class FooG {
  int size;
  float * data;
public:
  FooG(int N): size(N) {
    data = new float[N];
  }
  ~FooG() {
    delete[] data;
  }
};
```

destructor

- Whenever there's a "new", there must be a "delete"
- "new" in constructors are cleaned up in the destructor
  - destructor is called when variable goes out-of-scope or when its deleted

scope defined by {}

```
{
  FooG f(10);
  FooG * pf = new FooG(10);
  delete pf;  // *pf destructor (delete)
}  // f destructor at } (out-of-scope)
```

Classes - const function

```
class FooH {
public:
  void sayHi1() {
    cout << "HI\n" << endl;
  }
  void sayHi2() const {
    cout << "HI\n" << endl;
  }
};
```

```
FooH f;
f.sayHi1();  // OK
f.sayHi2();  // OK
const FooH cf;
cf.sayHi1(); // NOT ALLOWED
cf.sayHi2(); // OK
```

use case for const functions

- "const" after function signature means that "this" will not change when function is called
  - allows you to call things on variables of type "const FooH cf;"
- This would not compile:

```
class FooHB {
  int a;
public:
  void bad() const {
    a = 1; // NOT ALLOWED
  }
};
```

Classes - return (!) by reference

```
class FooA {
  float data[1000];
public:
  float& get(int i) {
    return data[i];
  }
  void sayHi() const {
    cout << "HI\n" << endl;
  }
};

class FooFooA {
  FooA f;
public:
  const FooA& getf() {
    return f;
  }
};
```

- Use case 1: change class variables

```
FooA f;
f.get(1) = 10;
```

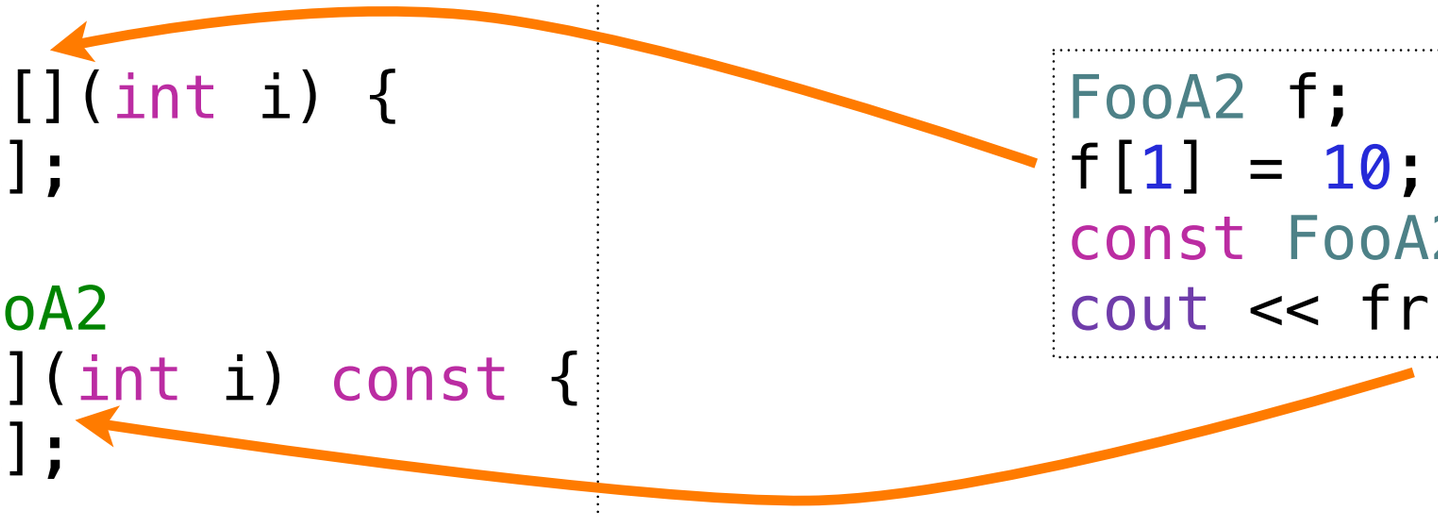
- Use case 2: return reference instead of copy of object
  - useful if object has lots of data (expensive to copy)
  - “const” makes sure we get unchangeable reference

```
FooFooA ff;
ff.getf().sayHi();
```

Classes - return (!) by reference with [] operator

```
class FooA2 {
  float data[1000];
public:
  // for FooA2
  float& operator[](int i) {
    return data[i];
  }
  // for const FooA2
  float operator[](int i) const {
    return data[i];
  }
};
```

```
FooA2 f;
f[1] = 10;
const FooA2& fr = f;
cout << fr[1] << endl;
```



- Using operator overloading for a more elegant “get” method
- operatorX can be used to override behavior of C++ operator X

Classes - copying data

```
class FooC {
  int a;
public:
  FooC() { a = 1; }
  FooC(const FooC& other): a(other.a) { }
  FooC& operator=(const FooC& other) {
    // check self-assignment
    if (&other == this) return *this;
    // copy fields
    a = other.a;
    // return required by C++ common practices...
    return *this;
  }
};
```

default constructor

copy constructor

assignment op.

```
FooC fc;
FooC fc2(fc);
FooC fc3 = fc;
fc3 = fc;
```

- Copy constructor: construct object as a copy of other one
- Assignment operator: overwrite object with data from other one
- C++ provides default copy constructor and assignment operator that do exactly what the ones here do
  - important in combination with pointers and dynamic data allocation (see Container)

Examples (codes will be put on webpage)

- **test.cpp**: mini test code with “int main()” for simple tests
  - pointers, references, const
- **commandline.cpp**: “int main(int argc, char ** argv)”
  - argc is number of command line tokens (space-delimited strings)
    (e.g. “./a.out -c -v bla” has argc = 4 tokens “./a.out”, “-c”, “-v” and “bla”)
  - argv[i] returns the token as a C string (char*) (i = 1..argc)
  - can also use “char * argv[]” instead of “char ** argv” and add “const” everywhere
- **container.cpp**: Container class
  - constructors, initialization lists, operators, function calls

Code layout

- **Main** program (commonly main.cpp)
  - contains main function (“int main()” or “int main(int argc, const char * argv[])”)
- Header files ***.h**
  - declare classes and functions <= *HINT: usually enough to understand what it does*
- Source files ***.cpp**
  - define functions (and global data) <= *HINT: only look at it if really, really necessary*
- Simple compilation of codes with multiple cpp files:
  - “g++ *.cpp” => generates executable “a.out” => execute with “./a.out”
- Advanced compilation: Makefile (provided by us in exercises)
  - compile with “make”
- Example: Container folder (equivalent to container.cpp)

Common coding issues

- I am not sure, what code ... does. How can I find out?
  - FIRST: do you really need to know? If not, forget it and move on...
  - meaning of C++ commands: check www.cppreference.com or www.cplusplus.com
  - custom commands: check header, comments and use common sense...
- Within a larger code: I am not sure if my addition ... works.
  - Test it! Either add code and see what happens (carefully) or (often better) write a **mini test code** to try it out.
- I am not sure how to solve a seemingly common problem.
  - Check the Internet if someone has solved it before. **Don't reinvent the wheel**.
    - But: don't copy blindly! Test with a small example whether it works for you.
  - E.g. reading in a file:
    - check tutorials: http://www.cplusplus.com/doc/tutorial/files/
    - look at solutions, find the one that suits your problem and adapt it to your needs