ETH zürich

401-0435-00L Computational Methods for Engineering Applications

Introduction

Eric Strauch
strauche@ethz.ch

D MATH

ETH zürich

Course Organisation

Teaching

- Lecture: Tuesday, 16-18, via zoom
- Exercise Classes: Thursday, 10.15-12, in presence or via zoom
  - inscribe accordingly

- Course Material on Moodle: **401-0435-00L Computational Methods for Engineering Applications HS2020**

Grading

- Written Session Exam in 2021 (Session: 25. Jan. 2021 19. Feb. 2021)
  - 180 mins
  - 4 pages personal summary (2 sheets) (for details consult course catalogue)
  - 100 pts
- Bonus: Two Projects featuring “Core problems”
  - Defence in interview with your assistant
  - 20 pts
- Max. grade can be achieved without the bonus
  - i.e. bonus pts are added to your score in the final exam

Exercises

- 6 exercise sets
  - Not graded
  - Calculations (derivations) and coding (C++)
  - Very beneficial as preparation for projects and exam!
- 2 Projects (Bonus for Core problems)
  - Due by end of October and 1st week of December
  - Published ~ 1 month prior
- Solutions will be published on Moodle

Tutorials

- Theory and concepts
- Introduction to exercise sets and projects

} Recorded

- Assistance for solving the exercises and projects
  - Or questions regarding the theory
- The tutorials are for you!
- Help me improve and shape the sessions!

ETH zürich

C++ Recap

Basics

```
double x = 5;
std::cout << x << std::endl;
```

- **Output:**

Basics

```
double x = 5;
std::cout << x << std::endl;
```

- **Output:**
  5

Basics

```
double x = 5;
std::cout << &x << std::endl;
```

- **Output:**

Basics

```
double x = 5;
std::cout << &x << std::endl;
```

- **Output:**
  0x62fde0

Basics

```
std::cout << x << std::endl;    //output value

std::cout << &x << std::endl;   //output address
```

Basics


```
double x = 5;
double y = x;
++y;
std::cout << x << std::endl;
std::cout << y << std::endl;
```

- **Output:**

Basics

```
double x = 5;
double y = x;
++y;
std::cout << x << std::endl;
std::cout << y << std::endl;
```

- **Output:**
  5
  6

Basics

```
double x = 5;
double y = x;                          //create copy of x
++y;                                   //increment copy
std::cout << x << std::endl;           //5
std::cout << y << std::endl;           //6
```

- **Output:**
  5
  6

Understanding references

```
double x = 5;
double &y = x;
++y;
std::cout << x << std::endl;
std::cout << y << std::endl;
```

- **Output:**

Understanding references

```
double x = 5;
double &y = x;
++y;
std::cout << x << std::endl;
std::cout << y << std::endl;
```

- **Output:**
  6
  6

Understanding references

```
double x = 5;
double &y = x;  //create alias (reference) to x named y
++y;            //increment reference
std::cout << x << std::endl;
std::cout << y << std::endl;
```

- **Output:**
  6
  6

Understanding references


```
void square_1(double x){
    x *= x;
}
```

```
double x = 5;
square_1(x);
std::cout << x << std::endl;
```

- **Output:**

Understanding references

```
void square_1(double x){
    x *= x;
}
```

```
double x = 5;
square_1(x);
std::cout << x << std::endl;
```

- **Output:**
  5

Understanding references

```
void square_1(double x){     //PASS BY VALUE -> local copy x of argument is made
    x *= x;                  //square the local copy x
}                            //end of scope -> local copy x is destroyed
```

```
double x = 5;
square_1(x);
std::cout << x << std::endl;
```

- **Output:**
  5

Understanding references

```
void square_2(double &x){
    x *= x;
}
```

```
double x = 5;
square_2(x);
std::cout << x << std::endl;
```

- **Output:**

Understanding references

```
void square_2(double &x){
    x *= x;
}
```

```
double x = 5;
square_2(x);
std::cout << x << std::endl;
```

- **Output:**
  25

Understanding references

```
void square_2(double &x){     //PASS BY REFERENCE -> local reference(/alias) x of argument is made
    x *= x;                   //now the original argument - variable is squared
}                             //end of scope -> local reference x is destroyed
```

```
double x = 5;
square_2(x);
std::cout << x << std::endl;
```

- **Output:**
  25

Understanding references

```
double square_3(double x){
    x += x;
    return x;
}

//make pass by value work
//change return value to double, and save result in new variable
double y = square_3(x);

//problem: A lot of copies -> bad memory management! (imagine vectors size 1b)
```

Understanding pointers


```
double x = 5;
double* ptr = &x;

++ptr;
std::cout << x << std::endl;
```

- **Output:**

Understanding pointers

```
double x = 5;
double* ptr = &x;

++ptr;
std::cout << x << std::endl;
```

- **Output:**
  5

Understanding pointers

```
double x = 5;
double* ptr = &x;

std::cout << ptr << std::endl;
```

- **Output:**

Understanding pointers

```
double x = 5;
double* ptr = &x;

std::cout << ptr << std::endl;
```

- **Output:**
  0x7ffeefbff4e8

Understanding pointers

```
void deref(){
    double x = 5;
    double* ptr = &x;

    std::cout << *ptr << std::endl;
}
```

```
deref();
```

Understanding pointers

```
void deref(){
    double x = 5;
    double* ptr = &x;

    std::cout << *ptr << std::endl;
}
```

```
deref();
```

- **Output:**
  5

Understanding pointers

```
void deref(){
    double x = 5;
    double* ptr = &x;

    ++(*ptr);
    std::cout << x << std::endl;
}

deref();
```

- **Output:**

Understanding pointers

```
void deref(){
    double x = 5;
    double* ptr = &x;

    ++(*ptr);
    std::cout << x << std::endl;
}

deref();
```

- **Output:**
  6

Understanding pointers


```
double* increment(){
    double x = 5;
    double* ptr = &x;

    ++(*ptr);
    return ptr;
}
```

- **Does this make sense?**

Understanding pointers

```
double* increment(){                    // Where's the problem?
    double x = 5;                       // Dangling pointer: double x = 5 is destroyed (end of scope)
    double* ptr = &x;                   // return ptr points to free memory -> undefined behaviour

    ++(*ptr);                           // key: always know where a pointer points to
    return ptr;                         // s.t. it doesn't point to a ghost variable
}
```

Dynamic memory

```
double* get_vec_s(){
    double x[3];
    return &x[0];    //invalid reference -> array gets destroyed
}
```

- How can we achieve that the object lives beyond the scope?
  - without declaring everything in main

Dynamic memory

```
double* get_vec(){
    double* x = new double[3];    //new allocation
    return x;
}                                 //array survives
```

- What's the consequence?

Dynamic memory

```
double* get_vec(){
    double* x = new double[3];    //new allocation
    return x;
}                                 //array survives
```

```
double* container = get_vec();

delete[] container;               //each new requires a corresponding delete (memory management)
container = nullptr;
```

Contact

Eric Strauch
strauche@ethz.ch

CMEA II Tutorial: Thu 10.15-12.00
Register according to your preferences

Course Material on Moodle

Thank you for your attention!