

# Object Oriented Scientific Programming in C++ (WI4771TU)

Matthias Möller  
Numerical Analysis

# Passing arguments – by value

- Pass by value `int function(int data)`
  - Function receives a copy of the original data
  - For large data a huge amount of temporal memory is needed
  - Modifications of data inside the function have no effect outside
  - Only one parameter can be changed at a time via return value
- Recommendation: I would use pass-by-value for passing fundamental data types and enumerators (later), e.g.  
`void printDayOfTheWeek(int day) { ... }`

# Passing arguments – by reference

- Pass by reference `void function(int& data)`
  - Function receives a reference to the original data
  - No copy of data is created (better performance for large data)
  - Modifications of data inside the function have effect outside
  - Multiple parameters can be changed at a time  
`void function (int& data1, int&data2, int&data3)`
  - To prevent modification of data but still enable efficient passing of arguments use `void function(const int& data)`
- Recommendation: I would use pass-by-reference in most cases using `const` to prevent data modification

# Passing arguments – by address

- Pass by address `void function(int* data)`
  - Function receives the address where data is stored in memory
  - The address is passed by value, so you cannot change the address but you can change the dereferenced data behind it
  - No copy of `data` is created (better performance for large data)
  - Modifications of `data` inside the function have effect outside
  - To prevent modification of `data` but still enable efficient passing of arguments use `void function(const int* data)`
- Recommendation: I would use pass-by-address only for build-in arrays using `const` to prevent data modification

# Passing arguments

- Example: Compute the sum of the entries of an array

```
double sum(const int* array, int length)
{
    double s = 0;
    for (auto i=0; i<length; i++)
        s += array[i];
    return s;
}
```

```
int array[5] = { 1, 2, 3, 4, 5 };
std::cout << sum(array, 5) std::endl;
```

# Task: Dot product

- Write a C++ code that computes the dot product

$$a \cdot b = \sum_{i=1}^n a_i b_i$$

of two vectors  $a=[a_1, a_2, \dots, a_n]$  and  $b=[b_1, b_2, \dots, b_n]$  and terminates if the two vectors have different length.

# Dot product function

- The main functionality without any fail-safe checks

```
double dot_product(const double* a, int n,  
                   const double* b, int m)  
{  
    double d=0;  
    for (auto i=0; i<n; i++)  
        d += a[i]*b[i];  
    return d;  
}
```

# C++ Assert

- C++ provides a general solution to ensure that prerequisites are satisfied and to gracefully exit otherwise

```
#include <cassert>
```

```
double dot_product(const double* a, int n,  
                   const double* b, int m)
```

```
{
```

```
    assert(n==m); // gracefully exit if n != m
```

```
    ... // compute dot product
```

```
}
```

- Interface: `void assert(int expression)`



# C++ Assert, cont'd

- All assert checks can be disabled explicitly by defining **NDEBUG** **before** the inclusion of the `cassert` header file

```
#define NDEBUG
#include <cassert>
double dot_product(const double* a, int n,
                  const double* b, int m)
{
    assert(n==m); // gracefully exit if n != m
    ... // compute dot product
}
```

- C++ assert can be very helpful for debugging but it should not be used as a general check for prerequisites!

# CMake build types

- CMake provides several default build modes
  - Debug: debug build with NDEBUG undefined
  - Release: release build with NDEBUG defined
  - RelWithDebInfo: release build(+debug info) w/NDEBUG defined
  - MinSizeRel: minimal size release build with NDEBUG defined
- Specify build mode when calling cmake

```
$> cmake -DCMAKE_BUILD_TYPE=Debug ../wi4771tu.2016.git
```

```
$> cmake -DCMAKE_BUILD_TYPE=Release ../wi4771tu.2016.git
```

# C++ Exceptions

- Exceptions are a more elegant way to handle failures
- Exceptions allow the user to react on failures instead of just gracefully exiting the program

```
#include <exception>
```

```
double dot_product(const double* a, int n,  
                  const double* b, int m)
```

```
{
```

```
    if(n!=m) throw „Vectors have different size“;
```

```
    ...
```

```
}
```

# Exceptions, cont'd

- Function that throws an exception is called in a try-block

```
#include <exception>
```

```
try
```

```
{
```

```
    dot_product(x, 5, y, 5); // can throw exception
```

```
} catch
```

```
{
```

```
    // This will be done if exception is thrown
```

```
    std::cerr << msg << std::endl;
```

```
}
```

# Exceptions, cont'd

- Exception is caught (with appropriate failure handling)

```
#include <exception>
try
{
    dot_product(x, 5, y, 4); // will throw exception
} catch
{
    // This will be done if exception is thrown
    std::cerr << msg << std::endl;
}
```

# Dot product improved

- First version of dot product with exception

```
#include <exception>
double dot_product(const double* a, int n,
                  const double* b, int m)
{
    if(n!=m) throw „Vectors have different length“;
    double d=0;
    for (auto i=0; i<n; i++)
        d += a[i]*b[i];
    return d;
}
```

# Dot product improved, cont'd

- First version of dot product is still not fully fail-safe

```
int main()
{
    double x[5] = { 1, 2, 3, 4, 5 };
    double y[4] = { 1, 2, 3, 4 };
    try {
        double d=dot_product(x, 5, y, 5);
    } catch {
        std::cerr << msg << std::endl;
    }
}
```

- It would be much better if x and y know about their length!

# Object Oriented Programming

- Main idea of OOP is to bundle data (e.g. array) and functionality (e.g. length) into a struct or class
- Components of a **struct** are public (=can be accessed from outside the struct) by default
- Components of a **class** are private (=cannot be accessed from outside the class) by default
- Components of a struct/class are **attributes** and **member functions** (=methods)



# Class vs. struct

- struct Vector  
{  
  public:  
    // default  
    double\* array;  
    int length;  
  private:  
}

- class Vector  
{  
  private:  
    // default  
  public:  
    double\* array;  
    int length;  
}

# Dot product with Vector

- Second version of dot product using Vector class or struct

```
#include <exception>
double dot_product(const Vector& a, const Vector& b)
{
    if(a.length!=b.length) throw „Vectors have ...“;
    double d=0;
    for (auto i=0; i<a.length; i++)
        d += a.array[i]*b.array[i];
    return d;
}
```

- Access member of a struct/class by dot-notation („.“)

# Dot product with Vector, cont'd

- ```
int main()
{
    Vector x,y;
    x.array = new double[5] {1,2,3,4,5}; x.length = 5;
    y.array = new double[5] {1,2,3,4,5}; y.length = 5;
    try {
        double d = dot_product(x, y);
    } catch {
        std::cerr << msg << std::endl;
    }
    delete[] x.array; x.length = 0;
    delete[] y.array; y.length = 0;
}
```

# Dot product with Vector, cont'd

- It is still possible to initialise `x.length` by the wrong value  
`x.array = new double[5] {1,2,3,4,5}; x.length = 4;`
- The main function is not very readable due to the lengthy declaration, initialisation and deletion of data
- OOP solution:
  - Constructor(s): **method** to construct a new **Vector object**
  - Destructor: **method** to destruct an existing **Vector object**

# Constructor

- The constructor is called each time a new Vector object (=instance of the class Vector) is created

```
class Vector
{
    // Default constructor
    Vector()
    {
        array = nullptr;
        length = 0;
    }
}
```

# Constructor, cont'd

- A class can have multiple constructors if they have a different interface (=different parameters)

```
class Vector
{
    // Constructor
    Vector(int len)
    {
        array = new double[len];
        length = len;
    }
}
```

# Constructor, cont'd

- What if a parameter has the same name as an attribute?

```
class Vector
{
    Vector(int length)
    {
        array = new double[length];
        // this pointer refers to the object itself, hence
        // this->length is the attribute and
        // length is the parameter
        this->length = length;
    }
    ...
}
```

# Destructor

- Destructor is called implicitly at the end of the lifetime of a Vector object, e.g., at the end of its scope

```
class Vector
{
    // Destructor (and there can be only one!)
    ~Vector()
    {
        delete[] array;
        length = 0;
    }
}
```



# Constructor/destructor

- ```
int main()
{
    Vector x; // Default constructor is called
    {
        Vector y(5); // Constructor is called
        // Destructor is called for Vector y
    }
    // Destructor is called for Vector x
}
```

# Uniform initialization constructors (C++11)

- But now, the handy uniform initialisation no longer works

```
Vector x(5); // use constructor
for (auto i=0; i<x.length; i++)
    x.array[i] = i;
```

- C++11 solution: initialiser lists (seems like magic right now)

```
Vector(std::initializer_list<double> list)
{
    length = (int)list.size();
    array = new double[length];
    std::uninitialized_copy(list.begin(),
                           list.end(), array);
}
```

# Dot product – close to perfection

- Third version of dot product using Vector class with uniform initialization constructor (C++11) and exceptions

```
int main()
{
    Vector x = { 1, 2, 3, 4, 5 };
    Vector y = { 2, 4, 6, 8, 10 };
    try {
        double dot_product(x, y);
    } catch {
        std::cerr << msg << std::endl;
    }
}
```

# Delegate constructor (C++11)

- `Vector(int length)`

```
{  
    array = new double[length];  
    this->length = length;  
}
```

```
Vector(std::initializer_list<double> list)
```

```
{  
    length = (int)list.size();  
    array = new double[length];  
    std::uninitialized_copy(list.begin(),  
                           list.end(), array);  
}
```

# Delegate constructor (C++11), cont'd

- Delegating constructors delegate part of the work to another constructor of the same or another class

```
Vector(int length)
: length(length) // initialise integer length to length
{
    array = new double[length];
}
```

- Here, delegation is not really helpful but more a question of coding style (e.g., some programmers use delegation in all situation where this is technically possible)

# Delegate constructor (C++11), cont'd

- Delegating constructors delegate part of the work to another constructor of the same or another class

```
Vector(std::initializer_list<double> list)
: Vector((int)list.size())
{
    std::uninitialized_copy(list.begin(),
                           list.end(), array);
}
```

- Here, delegation is really helpful since it reduces duplicate code (think: new attributes are added to Vector, then only one constructor has to be extended)

# Function -> member function

- Function that computes the sum of a Vector

```
double sum(const Vector& a)
{
    double s = 0;
    for (auto i=0; i<a.length; i++)
        s += a.array[i];
    return s;
}
```

- This is not really OOP-style!

```
int main()
{Vector x={1,2,3}; std::cout << sum(x) << std::endl;}
```

# Function -> member function, cont'd

- Vector object computes its sum (itself!)

```
class Vector
{
public:
    double sum()
    {
        double s = 0;
        for (auto i=0; i<length; i++)
            s += array[i];
        return s;
    }
}
```



# Function -> member function, cont'd

- This is good OOP-style

```
int main()
{
    Vector x = {1,2,3};
    std::cout << x.sum() << std::endl;
}
```

- Recommendation: prefer class/struct member functions over external ones whenever this is possible
- Can we implement the dot product as a member function?

# Function -> member function, cont'd

- ```
class Vector
{
public:
    double dot_product(const Vector& other)
    {
        if(length!=other.length) throw „Vectors have ...“;
        double d=0;
        for (auto i=0; i<length; i++)
            d += array[i]*other.array[i];
        return d;
    }
}
```

# Function -> member function, cont'd

- ```
int main()
{
    Vector x = {1,2,3}; Vector y = {2,4,6};
    std::cout << x.dot_product(y) << std::endl;
    std::cout << y.dot_product(x) << std::endl;
}
```
- Formally, the dot product is an operation between two Vector objects and not a member function of one Vector object the needs another Vector object for calculation

# Operator overloading

- C++ allows to overload (=redefine) the standard operators
  - Unary operators: ++a, a++, --a, a--, ~a, !a
  - Binary operators: a+b, a-b, a\*b, a/b
  - Comparison operators: a==b, a!=b, a<b, a<=b, a>b, a>=b
- Interfaces:  
`<returntype> operator<OP>() const`  
`<returntype> operator<OP>(const Vector& other) const`

# Operator overloading, cont'd

- Implementation of dot product as overloaded \*-operator

```
class Vector
{
public:
    double operator*(const Vector& other) const
    {
        if(length!=other.length) throw „Vectors have ...“;
        double d=0;
        for (auto i=0; i<length; i++)
            d += array[i]*other.array[i];
        return d;
    }
}
```

# Operator overloading, cont'd

- ```
int main()
{
    Vector x = {1,2,3}; Vector y = {2,4,6};
    std::cout << x*y << std::endl;
    std::cout << y*x << std::endl;
}
```
- Now, the dot product is implemented as \*-operation that maps two Vector objects to a scalar value

# Addition by operator overloading

- Implementation of addition as overloaded +-operator

```
Vector& operator+(const Vector& other) const
{
    if(length!=other.length) throw „Vectors have ...“;
    for (auto i=0; i<length; i++)
        array[i] += other.array[i];
    return *this;
}
```

- Usage: `Vector x, y; x = x+y;`

# Assignment by operator overloading

- Implementation of assignment as overloaded -=operator

```
Vector& operator=(Vector& other) const
```

```
{
```

```
    if(this != &other)
```

```
    {
```

```
        // Exchange resources between *this and other
```

```
        swap(other);
```

```
    }
```

```
    return *this;
```

```
} // destructor of other is called to release resources
```

```
    // formerly held by *this
```