What you always wanted to know…
but never dared to ask:

C++

Prof. Dr. Elmar Eisemann
Computer Graphics and Visualization
TU Delft

C++ Basics

Code Organization

- A C++ Project is spread into several files
- Only one file contains a main() function
- Before using a function/class
  - Name
  - Return type
  - Amount and types of parameters

have to be KNOWN, but NOT implemented!

These definitions can be in separate files (so-called header files), while the implementations are usually in “cpp” files.

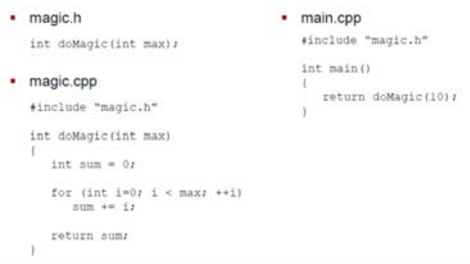
Code Organization

Definitions in a “.h” file (header)
Implementations in a “.cpp” file

To use a function or class in local files:
#include “name”

To use header from a system repository:
#include <name>

Simple example



- magic.h

```
int doMagic(int max);
```

- magic.cpp

```
#include "magic.h"

int doMagic(int max)
{
   int sum = 0;

   for (int i=0; i < max; ++i)
      sum += i;

   return sum;
}
```

- main.cpp

```
#include "magic.h"

int main()
{
   return doMagic(10);
}
```

ATTENTION

- Every definition should only appear ONCE

- Avoid double inclusion of “.h” files
- Add

#ifndef MYMAGIC_IS_UNIQUE_NAME
#define MYMAGIC_IS_UNIQUE_NAME
….
#endif

MYMAGIC_AND_UNIQUE_NAME can be any unique name, e.g., HAPPY_HIPPO could be used as well as long as no other file uses it
Alternatively, in most compilers (Visual C++) you can use “#pragma once”

Namespaces

- C++ also introduces namespaces
- Imagine you include two files that define the same function with different functionalities, to distinguish both, surround definitions by

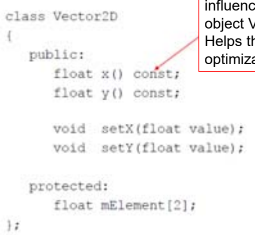
```
namespace NiceNAME {
        […]
        void myFunc() {[…]}
}
```

- To access function use NiceNAME::myFunc()

Class Definitions

```
Class A
{
        public:
        //define public functions/constructors/operators
        protected:
        //protected functions/constructors/operators
        private:
        //private functions/constructors/operators
};
```

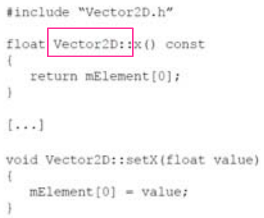
Example: A 2D Vector Class



```
class Vector2D
{
   public:
      float x() const;
      float y() const;

      void  setX(float value);
      void  setY(float value);

   protected:
      float mElement[2];
};
```

Shows the compiler that these functions will not influence the variables in the object Vector2D.
Helps the compiler for optimizations!

- In the Vector2D.cpp file, reference the class
- Example:

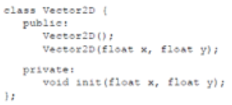


```
#include "Vector2D.h"

float Vector2D::x() const
{
   return mElement[0];
}

[...]

void Vector2D::setX(float value)
{
   mElement[0] = value;
}
```

Constructors

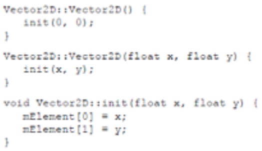
- To define an object

Definition



```
class Vector2D {
   public:
      Vector2D();
      Vector2D(float x, float y);

   private:
      void init(float x, float y);
};
```

Implementation :

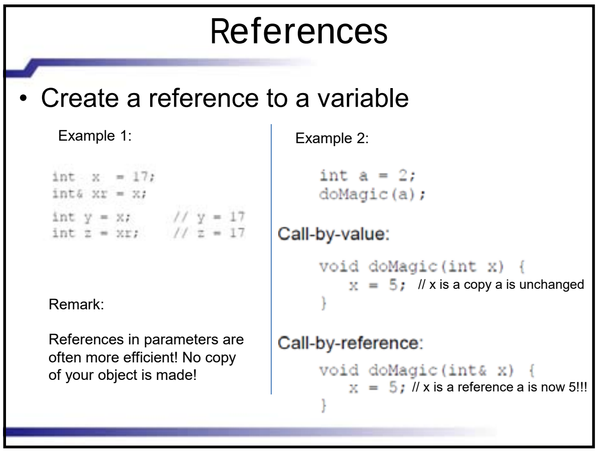


```
Vector2D::Vector2D() {
   init(0, 0);
}

Vector2D::Vector2D(float x, float y) {
   init(x, y);
}

void Vector2D::init(float x, float y) {
   mElement[0] = x;
   mElement[1] = y;
}
```

Destructor

- Cleanup function when object is destroyed

```
Vector2D::~Vector2D()
{
        //in this case: no memory to release,
        //hence no cleanup needed
}
```



References

- Create a reference to a variable

Example 1:

```
int  x  = 17;
int& xr = x;

int y = x;      // y = 17
int z = xr;     // z = 17
```

Remark:

References in parameters are often more efficient! No copy of your object is made!

Example 2:

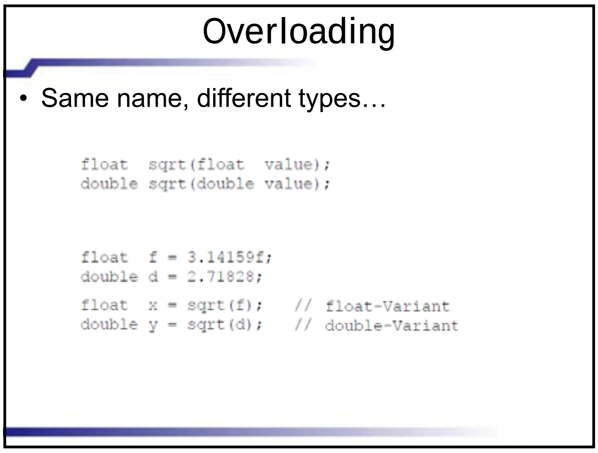
```
int a = 2;
doMagic(a);
```

Call-by-value:

```
void doMagic(int x) {
    x = 5;  // x is a copy a is unchanged
}
```

Call-by-reference:

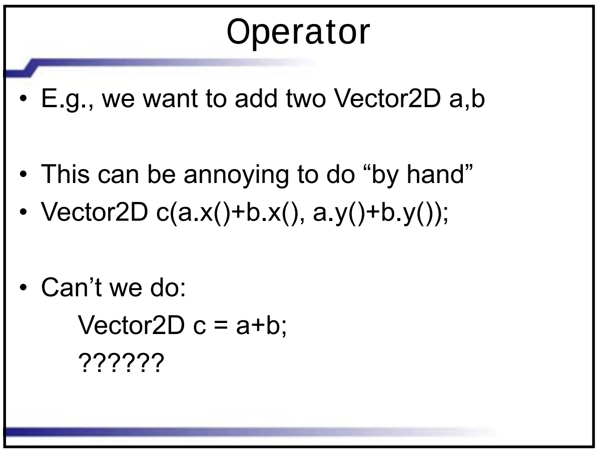
```
void doMagic(int& x) {
    x = 5; // x is a reference a is now 5!!!
}
```



Overloading

- Same name, different types…

```
float  sqrt(float  value);
double sqrt(double value);


float  f = 3.14159f;
double d = 2.71828;

float  x = sqrt(f);   // float-Variant
double y = sqrt(d);   // double-Variant
```



Operator

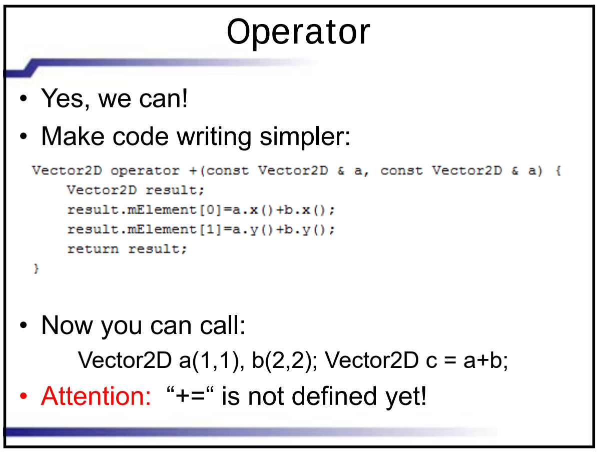
- E.g., we want to add two Vector2D a,b

- This can be annoying to do “by hand”
- Vector2D c(a.x()+b.x(), a.y()+b.y());

- Can’t we do:

  Vector2D c = a+b;

  ??????



Operator

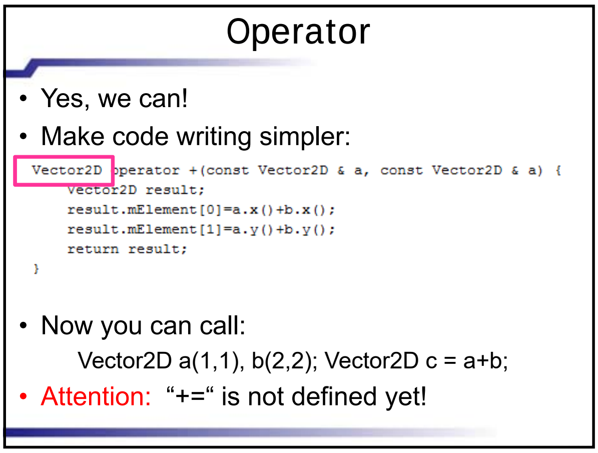
- Yes, we can!
- Make code writing simpler:

```
Vector2D operator +(const Vector2D & a, const Vector2D & a) {
    Vector2D result;
    result.mElement[0]=a.x()+b.x();
    result.mElement[1]=a.y()+b.y();
    return result;
}
```

- Now you can call:

  Vector2D a(1,1), b(2,2); Vector2D c = a+b;

- Attention: “+=“ is not defined yet!



Operator

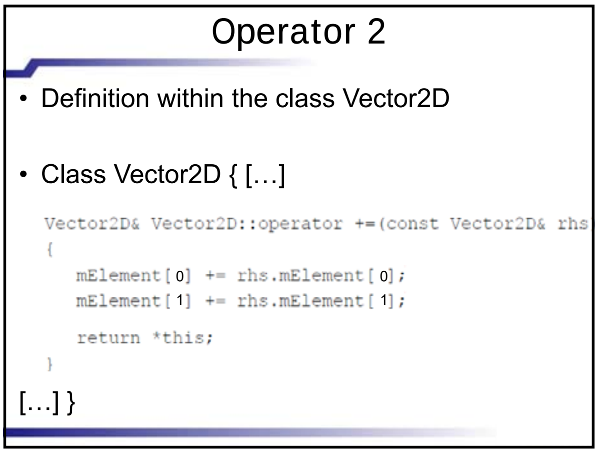
- Yes, we can!
- Make code writing simpler:

```
Vector2D operator +(const Vector2D & a, const Vector2D & a) {
    Vector2D result;
    result.mElement[0]=a.x()+b.x();
    result.mElement[1]=a.y()+b.y();
    return result;
}
```

- Now you can call:

  Vector2D a(1,1), b(2,2); Vector2D c = a+b;

- Attention: “+=“ is not defined yet!



Operator 2

- Definition within the class Vector2D

- Class Vector2D { […]

```
Vector2D& Vector2D::operator +=(const Vector2D& rhs
{
    mElement[0] += rhs.mElement[0];
    mElement[1] += rhs.mElement[1];

    return *this;
}
```

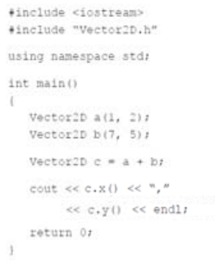
[…] }

Operators 3

- Remember, often you should consider many combinations!
- For example:

```
Vector2D operator* (float s, const Vector2D& v);

Vector2D operator* (const Vector2D& v, float s);
```

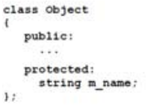
Example



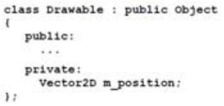
```
#include <iostream>
#include "Vector2D.h"

using namespace std;

int main()
{
   Vector2D a(1, 2);
   Vector2D b(7, 5);

   Vector2D c = a + b;

   cout << c.x() << ","
        << c.y() << endl;

   return 0;
}
```

Object-Oriented Programming

- Inherit from one (or multiple objects)



```
class Object
{
   public:
     ...
   protected:
     string m_name;
};
```



```
class Drawable : public Object
{
   public:
     ...
   private:
     Vector2D m_position;
};
```

- As well as virtual methods, abstract classes…

“static”

- “static” refers to the class itself, these functions and variables are shared among all objects of this class

```
class A
{ public:
        A(){++i};
        ~A(){--i};
        static int i;
};
int A::i=0;
```

The variable i will count the number of object instances of type class A.

Important Standard Class: Vector

- Tables:

```
#include <vector>
#include “Vector2D.h”
using namespace std;
…
vector<float> a; //has usually no entries
vector<float> b(10);// will have 10 entries
a.push_back(2.3f);
a.size(); // now 1
a[0] = 2.4f;
a[1]= 3.4f;     // BIG NO NO !!! In debug it will complain,
                //release accepts this!!! Oh no…
```

Important Objects

- Nice: Customize your vectors…

```
vector<Vector2D> points;
vector<vector<int> > test(100, 100);
```

Important Objects

- Nice: Customize your vectors…

```
vector<Vector2D> points;
vector<vector<int> > test(100, 100);
```

ATTENTION

Never do something like this:

```
int n=3;
float pos[n];
//might compile in gcc,
//not always in vcc…
```

USE vectors instead!!! Otherwise:

Only exception: small tables of constant size…
e.g., int pos[3]; …but then still be careful!

One more remark on “new”

In C++ there is also a “new” command, BUT
Vector2D t= new Vector2D(); // will not compile!

“new” allocates memory and returns a pointer
Vector2D * t= new Vector2D(); // will compile

In this case you also need to “delete” afterwards
-> DON’T USE “new”!!! DON’T USE pointers!
Inefficient, but if you like the java way:

```
Vector2D t=Vector2D();
```

Before I forget…

- Next week:

NO LECTURE &
NO INSTRUCTION
on the 8th of May!!!

EXERCISES take place
on the 7th of May!!!

Novice Programmer
DING!
I CAN'T BELIEVE IT WORKED FIRST TIME!
Experienced Programmer
DING!
I CAN'T BELIEVE IT WORKED FIRST TIME...

Time for action!