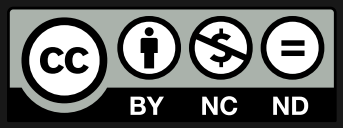
Math 4997-1

Lecture 5: Operator overloading and structuring programs

Patrick Diehl 

https://www.cct.lsu.edu/~pdiehl/teaching/2020/4997/

This work is licensed under a Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International” license.



Reminder

Operator overloading

Structure of code

- Header files
- Class types

CMake

Summary

Reminder

Lecture 4

What you should know from last lecture

- *N*-Body simulations
- Struct
- Generic programming (Templates)

Operator overloading

Example

Vector

```
template<typename T>
struct vector {
T x;
T y;
T z;
};
```

Addition of two vectors

```
vector<double> a;
vector<double> b;
std::cout << a + b << std::endl;
```

Example

Vector

```
template<typename T>
struct vector {
T x;
T y;
T z;
};
```

Addition of two vectors

```
vector<double> a;
vector<double> b;
std::cout << a + b << std::endl;
```

Compilation error

```
error: no match for ‘operator’+
(operand types are ‘’vector and ‘’vector)
```

Operator overloading[1]

```
template<typename T>
struct vector {
T x;
T y;
T z;
// Overload the addition operator
vector<T> operator+(const vector<T> rhs){
return vector<T>( x + rhs.x, y + rhs.y, z + rhs.z );
}
};
```

Following operators can be overloaded

- 38 operators can be overloaded
- 40 operators can be overloaded, since C++ 20

[1] https://en.cppreference.com/w/cpp/language/operators

Can we compile now?

```
template<typename T>
struct vector {
T x;
T y;
T z;
// Overload the addition operator
vector<T> operator+(const vector<T> rhs){
return vector<T>( x + rhs.x, y + rhs.y, z + rhs.z );
}
};
```

Can we compile now?

```
template<typename T>
struct vector {
T x;
T y;
T z;
// Overload the addition operator
vector<T> operator+(const vector<T> rhs){
return vector<T>( x + rhs.x, y + rhs.y, z + rhs.z );
}
};
```

D'oh!

```
error: no match for ‘operator’<<
(operand types are ‘std::ostream
{aka std::basic_ostream’} and ‘’vector)
     std::cout << a + b << std::endl;
```

Overload the next parameter

```
template<typename T>
struct vector {
T x, y, z;
vector(T x, T y, T z) : x(x),y(y),z(z) {};
// Overload the addition operator
vector<T> operator+(const vector<T> rhs){
return vector<T>( x + rhs.x, y + rhs.y, z + rhs.z );
}
//Overload the output operator
friend ostream& operator
         <<(ostream& os, const vector<T>& vec)
{
    os << vec.x << " " << vec.y << " " <<  vec.z;
    return os;
}
};
```

We will have a closer look to `friend` in the next section.

Structure of code

Organization of code

C++ provides two fundamental ways to organize the code

- Functions or so-called subroutines, *e.g.* `double norm()`
- Data structures, *e.g.* `struct`

we have learned so far. A new opportunity is to split the code into different files to make all files *shorter* and separate the code by its *functionality*.

Let us look into header files[2] first and later at classes to do this.
More details [**?**, **?**].

[2] https://docs.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=vs-2019

Header file

- A common naming convention is that header files end with *.h* or *.hpp*, e.g. *average.h*
- We include them into our code by using

```
#include<average.h>
```

- Note the inclusions form the C++ standard library do not end with *.h* or *.hpp*

Example of the average.h file

```
// Utils for the vector container
namespace util {

}
```

Namespaces[3] `namespace` are used to avoid naming conflicts and structure in large projects.

[3] https://en.cppreference.com/w/cpp/language/namespace

Adding code to the header file

```
// Average of the elements of a vector
#include <vector>
#include <algorithm>

namespace util {

double average(std::vector<double> vec){
return std::accumulate(vec.begin(), vec.end(), 0.0f)
     / vec.size();
}

}
```

Usage

```
#include "average.h"
double res = util::average(vector);
```

Include guards

```
#ifndef UTIL_H // include guard
#define UTIL_H

// Average of the elements of a vector
#include <vector>
#include <algorithm>

namespace util {

double average(std::vector<double> vec){
return std::accumulate(vec.begin(), vec.end(), 0.0f)
    / vec.size();
}
}
#endif
```

Include gards avoid that functions or data structures are multiple defined. Short from: `#pragma once`

Remarks for header files

Following things are considered as good practice:

- Each header file provides exactly one functionality
- Each header file includes all its dependencies

Following things should not be in header files and be considered as bad practice:

- built-in type definitions at namespace or global scope
- non-inline function definitions
- non-const variable definitions
- aggregate definitions
- unnamed namespaces
- using directives

Compilation with header files

Folder structure

```
sources/
    main.cpp
includes/
    average.h
```

File main.cpp

```
#include<average.h>

int main(void){
    std::cout << util::average(vec) << std::endl;
}
```

Compilation

```
g++ -o main -I ../includes main.cpp
```

Definition of a `class` type[4]

```
class vector2 {

private:

double x , y , z;

public:

vector2(double x = 0, double y=0, double z=0)
    : x(x) , y(y) ,z(z) {}
double norm(){ return std::sqrt(x*x+y*y+z*z);}
};
```

Access specifier:

- `public` – members are accessible from outside the class
- `private` – members cannot be accessed from outside

[4] https://en.cppreference.com/w/cpp/language/classes

Definition of classes

```
class vector2 {
private:
double x , y , z;
public:
vector2(double x = 0, double y=0, double z=0)
        : x(x) , y(y) ,z(z) {}
double norm(){ return std::sqrt(x*x+y*y+z*z);}
};

int main()
{

    vector2 vec = vector2();

    return 0;
}
```

Structuring of classes

Header file (vector.h)

```
class vector2 {

private:

double x , y , z;

public:
vector2(double x = 0, double y=0, double z=0);

double norm();
};
```

In a header file the attributes and the member function of the class are defined.

Structuring of classes

Class file (vector.cpp)

```
#include "vector2.h"

vector2::vector2(double x, double y, double z)
{
    x = x; x = y; z = z;
}

double vector2::norm(){return std::sqrt(x*x+y*y+z*z)}
```

- In the cpp file the implementation of the members functions and the constructor is defined.
- The corresponding header file needs to be included.
- The header file has to been included to access the `public` member functions and attributes of the class.
- The class file needs to be compiled before it can be used.

Usage and compilation

```
#include "vector2.h"

int main()
{

    vector2 vec = vector2();

    return 0;
}
```

Compilation

```
g++ -c vector2.cpp
g++ main.cpp vector2.o
```

We do not want to do this for several files or?

CMake

CMake[5]

CMake is a cross-platform free and open-source software tool for managing the build process of software using a compiler-independent method. It supports directory hierarchies and applications that depend on multiple libraries. It is used in conjunction with native build environments such as Make, Ninja, Apple's Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system.

[5] https://cmake.org/

Compile a single cpp file

Content: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10.1)
project (hello_world)
add_executable(hello main.cpp)
```

Running cmake

```
mkdir build
cd build
cmake ..
make
./hello
```

Compiling a class file and a main file

Folder structure

```
.
|-- CMakeLists.txt
|-- build
|-- include
|   \-- vector2.h
\-- src
    |-- vector2.cpp
    \-- main.cpp
3 directories, 4 files
```

Corresponding CMakeLists.txt

```
project(directory_test)

#Bring the headers, such as Student.h into the project
include_directories(include)

#Manually adding all sources
#set(SOURCES src/main.cpp src/vector2.cpp)

#Adding sources easier
file(GLOB SOURCES "src/*.cpp")

add_executable(test ${SOURCES})
```

Summary

Summary

After this lecture, you should know

- Operator overloading
- Splitting class types in header and class files
- Building projects using CMake