

Chapter 4 - Classes

From struct to class

- Structs contain member-variables and member-functions. Without extra declaration all components of a struct are public.
- Upon creating an instance of a struct only memory is allocated, no initialization is done.
- There is a rudimentary assignment operator (=) available for structs, but no other operators such as +, -, *, ...
- Classes contain member-variables and -functions. Without extra declaration all components of the class are private.
- All classes must have a constructor and a destructor. A constructor specifies what has to be done when an object is created, a destructor takes care of cleaning up once an object reaches the end of its lifetime.
- For classes we will overload operators, i.e. define operators +, *, /, ... which can work with instances of the class.

A simple example

- These two codes do identical things...

```
struct particle{
public:
    void initialize();
private:
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//-----
int main(){
    particle p;

    p.initialize();
}
//-----
// All member variables to 0
void particle::initialize(){
    m = 0; q = 0; n=0;
    vx = 0; vz = 0; vy = 0;
    x = 0; y = 0; z = 0;
}
```

```
class particle{
public:
    particle();
private:
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//-----
int main(){
    particle p;
}
//-----
// Put all member variables to 0
particle::particle(){
    m = 0; q = 0; n=0;
    vx = 0; vz = 0; vy = 0;
    x = 0; y = 0; z = 0;
}
```

Constructors

- A constructor performs initialization of an object once it is being created.
- A class has to have at least one constructor, but may have more than this one (standard constructor, copy constructor, type-cast constructor,...).
- Constructors
 - ▶ are member functions of the class which have the same name as the class,
 - ▶ may have arguments (the default constructor has no argument),
 - ▶ have no return value (not even void).
- If no constructor at all is given, C++ will create a default constructor during compilation. It will be an empty function without any argument.
- As soon as we specify any constructor, C++ will not generate a default constructor.
- Three equivalent classes:

```
class A{  
public:  
    A();  
private:  
    double d;  
};  
  
A::A(){}
```

```
class A{  
public:  
    A(){}  
private:  
    double d;  
};
```

```
class A{  
private:  
    double d;  
};
```

Constructors

```
//-----  
class particle{  
public:  
    //Default constructor  
    particle();  
  
    //Standard constructor  
    particle(const double X, const double Y, const double Z,  
             const double VX, const double VY, const double VZ,  
             const double M, const double Q, const int N);  
  
private:  
    int n;  
    double x,y,z;  
    double vx,vy,vz;  
    double q,m;  
};  
//-----  
int main(){  
    // Calls default constructor  
    particle p1;  
  
    // Calls standard  
    particle p2(0, -1 , 0, -2, 0, 0, 1, -1, 1);  
}
```

```
//-----  
// Default constructor  
particle::particle(){  
    m = 0; q = 0; n=0;  
    vx = 0; vz = 0; vy = 0;  
    x = 0; y = 0; z = 0;  
}  
//-----  
// Standard constructor  
particle::particle(const double X, const double Y,  
                  const double Z,  
                  const double VX, const double VY,  
                  const double VZ,  
                  const double M, const double Q,  
                  const int N)  
{  
    x = X; y = Y; z = Z;  
    vx = VX; vy = VY; vz = VZ;  
    n = N;  
    q = Q;  
    m = M;  
}
```

Destructor

- Each class has to have a destructor which cleans up when the life time of an object ends
- There is always only one destructor per class
- The destructor is a member function of the class and has the name of the class with ~ as prefix
- The destructor has no argument and no return value
- When no destructor is given, C++ will generate one for the class. Like the automatically generated function it is just an empty function.

```
class A{
public:
    A(){d=0};
    A(const int D){d = D;}
private:
    double d;
};

int main(){
    A a;
    A b(1.0);
} // <- Life-time of a and b ends here
```

```
class A{
public:
    A(){d=0};
    A(const int D){d = D;}
    ~A(){}; // Destructor
private:
    double d;
};

int main(){
    A a;
    A b(1.0);
} // <- Life-time of a and b ends here
```

Destructor

- Custom destructors are very important when we dynamically allocate memory inside a constructor, because we have to free that memory once the object dies

```
class Vec{
public:
    Vec(const int size){
        p = new double[size];
    }
    ~Vec(){ delete[] p;} // Destructor
private:
    double* p;
};

int main(){
    // Vec v1; //Not valid, no default constructor in class
    Vec v2(10);
} // <- Life-time of v2 ends here
```

this - pointer

- In our examples we often have the case that we would like to name function parameters identically to the member variables. Up to now we circumvented the problem by using capital versions of the variable names for the parameters

```
// Standard constructor
particle::particle(const double X, const double Y, const double Z,
                  const double VX, const double VY, const double VZ,
                  const double M, const double Q, const int N)
{
    x = X; y = Y; z = Z;
    vx = VX; vy = VY; vz = VZ;
    n = N;
    q = Q;
    n = N;
}
```

- We may not use parameter names which are identical to member variable names, since then the parameter variables would shadow the member variables.

```
// Standard constructor
particle::particle(const double x,...)
{
    x = x;
}
```

Here x on both sides of = will be the parameter we passed to the function. We need a way to specify that the x on the left side is the member variable x.

this - pointer

- Inside each member function we have access to a pointer called `this`. The `this` pointer always points to the object to which the member function belongs. Now we have another way to clearly distinguish between member variables and parameters:

```
particle::particle(const double x, const double y, const double z,  
                  const double vx, const double vy, const double vz,  
                  const double m, const double q, const int n){  
    this->x = x; this->y = y; this->z = z;  
    this->vx = vx; this->vy = vy; this->vz = vz;  
    this->n = n; this->q = q; this->n = n;  
}
```

- The `this` pointer is not only available in constructors but in all member functions.
- We may use it everywhere, where we need to specify that we refer to a member variable or function that might be shadowed by a parameter otherwise.
- Specifically for constructors there is a second way to deal with the problem of common variable names, the *constructor initializer list*. More on this later.

const member functions

```
class A{
public:
    A(){};
    ~A(){};
    double get();
    void set(const double q);
private:
    double q;
};
//-----
double A::get(){
    return q;
}
//-----
void A::set(const double q){
    this->q = q;
}
//-----
int main(){
    const A a;

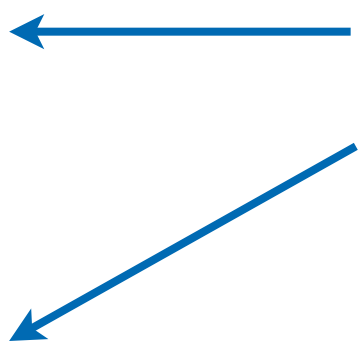
    // Both will not work:
    a.set(1);
    a.get();
}
```

- When we try to compile this code the compiler gives the error:
../ClassExample3.cxx:23: error: passing 'const A' as 'this' argument of 'void A::set(double)' discards qualifiers
../ClassExample3.cxx:24: error: passing 'const A' as 'this' argument of 'double A::get()' discards qualifiers
- a is a const object, i.e. it may never be changed after creation.
- Member functions of a might change the content of the member variables (set() does, get() does not)
- C++ will not check by itself whether a member function changes the member variables
- We have to specify which member functions do not change anything inside the object by adding const to the function declaration
- Only member functions which are declared as const can then be called from a const object.


const member functions

```
class A{
public:
    A(){};
    ~A(){};
    double get() const;
    void set(const double q);
private:
    double q;
};
//-----
double A::get() const{
    return q;
}
//-----
void A::set(const double q){
    this->q = q;
}
//-----
int main(){
    const A a;

    // set() will not work, since
    // a is const A:
    a.set(1);
    // But get() will work now:
    a.get();
}
```



const has been added to mark get() as a function which does not change the member variables when called



Now we may call a.get() since we guaranteed that get() leaves the members of a unchanged

const member functions

- A class may have two member functions with the same name and parameters, where one is the const version and the other is non-const.

```
#include <iostream>
using namespace std;

class A{
public:
    double get() const;
    double get();
private:
    double q;
};
//-----
double A::get() const{
    cout << "get() const" << endl;
    return q;
}
//-----
double A::get(){
    cout << "get()" << endl;
    return q;
}
//-----
void f(const A& c){
    c.get();
}
```

```
int main(){
    const A a;
    A b;

    a.get();
    b.get();
    f(a);
    f(b);
}
```

Output:

```
get() const
get()
get() const
get() const
```

Const member variables

- There might be cases where we want a member variable of a class to hold a value which should be constant.
- We have to distinguish two cases here:
 - ▶ All objects of the class should hold the same value in this member variable
Example: All objects of a class representing particles might need the value of the vacuum permittivity $\epsilon_0 = 8.85e-12$ stored into a member variable `eps0`. It is the same for all objects.
 - ▶ Each object has a member variable which is once initialized to a certain value and throughout the rest of the program this value should never be changed again.
Example: A particle object is given a certain value for the particle mass upon construction. This value shall never be changed again. Each particle object however may hold a different mass value.

static const member variables

- We can use static member variables to introduce a member variable to the class which is constant and carries *the same value for all objects of the class*.

```
class A{
public:
    A(){};
    static const int j;
private:
    static const double epsilon;
    static const int i=5;
};
//-----
const double A::epsilon=8.85e-12;
const int A::j=2;

int main(){
    A ant;
}
```

- Before const we write static. This tells the compiler that we will need memory to hold the value even if there is no instance of the class.
- Integer const variables may be assigned their value inside the class declaration.
- Other constant variables are set *outside* of the class using the scope-resolution operator :: to address the correct variable.
- Note that the definition of the const variable is also outside of any other function. They will not be global variables however, because the variables live inside the namespace of the class.

const member variables

- Goal: Have a member variable which is once given a value upon creation and then may never be changed again. Every instance of a class may hold a different value in this const variable.
- Problem: A C++ variable that is const must have a value when it is defined. We know the value only when we call the constructor of the class.

```
class A{  
public:  
    A(const double M);  
private:  
    const double m;  
};  
//-----  
A::A(const double M){  
    m = M;  
}  
//-----  
int main(){  
    A ant(1.0);  
}
```

```
../ClassExample3.cxx: In constructor 'A::A(double)':  
../ClassExample3.cxx:11: error:  
                        uninitialized member 'A::m' with 'const' type 'const double'  
../ClassExample3.cxx:12: error: 'M' was not declared in this scope
```

← This does not work since when the body of the constructor function opens with { the memory for m has already to be initialized.

We need a way to do `m=M` before the constructor body...

Constructor initializer list

- To initialize const member variables before the constructor of a class starts C++ provides the *constructor initializer list*.
- The *constructor initializer list* is placed after the parameters of the constructor, but before the opening { of the constructor body.

```
//-----  
class A{  
public:  
    A(const double M, const double N);  
private:  
    const double m;  
    const double n;  
};  
//-----  
A::A(const double M, const double N):m(M),n(N){  
}  
//-----  
int main(){  
    A ant(1.0, 0.5);  
}
```

- The statements variable1(variable2) in the list have the meaning variable1 = variable2;
- variable1 is always a variable of the class, variable2 is a variable from the constructors parameters list or a constant.
- Initializer lists can (and are) be used to initialize also non-const member variables.

Constructor initializer list

- The initializer list for constructors finally also allows us to use the same variable names for the constructor parameters and the member variables in a simple way.
- All three examples are equivalent:

```
class A{  
public:  
    A(const double M);  
private:  
    double m;  
};  
//-----  
A::A(const double M){  
    m = M;  
}
```

```
class A{  
public:  
    A(const double m);  
private:  
    double m;  
};  
//-----  
A::A(const double m){  
    this->m = m;  
}
```

```
class A{  
public:  
    A(const double m);  
private:  
    double m;  
};  
//-----  
A::A(const double m):m(m){  
}
```

Optional function arguments

- Functions might have parameters which have almost always the same value or in rare cases we might need an extra parameter when calling a function.
- It is possible to give function parameters default values that they will hold if no other value for this parameter is given.
- The default values are specified in the function declaration, but do not appear in the function definition.

```
#include <cmath>
//-----
double sqr(double x, double a=0.5);
//-----
int main(){
    double x1 = sqr(2);
    double x2 = sqr(2, 1.0/3.0);
}
//-----
double sqr(double x, double a){
    return pow(x,-a);
}
```

← `sqr` can be called with one or two arguments.
If called with only one argument, the value of `a` is the default 0.5.

Optional function arguments

- Optional parameter(s) must always be the last parameter(s) in the function header. How else could the compiler know which parameter we skipped?
- We can also use optional arguments in constructors for classes:

```
class matrix{  
public:  
    matrix(const int N, const int M, const std::string name, const bool init=true);  
}
```

```
int main(){  
    matrix M1(3,4,"M1", true);  
    matrix M2(3,4,"M2");  
    matrix M2(3,4,"M2", false);  
}
```

friends

- A class can declare functions outside of the class to be a friend.
- Friend functions have access to all class members (public/private/protected) but are not member functions.
- To declare a function friend of the class the keyword `friend` is attached before the function declaration in the class header.

```
class monster{
public:
    monster(){
        hungry = false;
        level = 0; }
    ~monster(){};
    friend void makeHungry(monster& m, const int level);
private:
    bool hungry;
    int level;
};

void makeHungry(monster& m, const int level){
    m.hungry = true;
    m.level = level;
}

int main(){
    monster godzilla;
    makeHungry(godzilla, 10);
}
```

friends

- Not only functions can be friends of classes but also other classes can become friends.

```
class A;  
class B;  
  
class A{  
    friend class B;  
public:  
    ...  
private:  
    ....  
};
```

- In this way all member functions of B have full access to members of A.
- Declaring B friend of A does not make A friend of B, i.e. A may not access all members of B (extra friend declaration has to be done in class B to achieve this).
- Use friends only where really necessary, because in this way we break up the concept of encapsulation.

Copy constructor

- The copy constructor creates a new instance by cloning an existent object.

```
class particle{
public:
    //Default constructor
    particle();

    //Standard constructor
    particle(const double X, const double Y, const double Z,
            const double VX, const double VY, const double VZ,
            const double M, const double Q, const int N);

    // Copy constructor
    particle(const particle& p);
private:
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//-----
particle::particle(const particle& p){
    n = p.n; x = p.x; y = p.y; y = p.z;
    // and so forth...
}
//-----
int main(){
    particle e(0, 0, 0, -1, 0, 0, 1, 1, 1);
    particle f(e);
}
```

- C++ will provide a copy constructor if we do not implement our own.
- Just as the assignment operator = C++ provides, the copy constructor will perform a *shallow copy*, i.e. bit-wise copy, of the member variables.
- In the case shown here our copy constructor is just the same as the one C++ would generate for us.

Copy constructor

- Again we have to be careful when the class contains a pointer member variable (or any other resource handle). Now we need copy constructor which does a *deep copy* generate a true copy which does not share any resources with the original object.

```
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v);  // Copy constructor
    ~Vec(){ delete[] p;} // Destructor
private:
    const int dim;
    double* p;
};
//-----
Vec::Vec(const int dim):dim(dim){
    p = new double[dim];
}
//-----
Vec::Vec(const Vec& v):dim(v.dim){
    p = new double[dim];
    for(int i=0; i<dim; i++) p[i] = v.p[i];
}
//-----
int main(){
    Vec v1(10);
    Vec v2(v1);
}
```

- Creating a true copy of an object requires resources (time, memory) and is thus to be avoided where possible.
- When passing objects to functions, check if only read access is required. Then pass only a const reference to the original.
- More sophisticated copy: Still share resources with the original object but do *reference counting*.

Overloading operators

- C++ allows us to give operators like `+=`, `-=`, `...`, `+`, `-`, `...`, `%`, `=`, `==`, `[]`, `()` a special meaning when used in the context of objects of a class. This is called *operator overloading*.
- Operator overloading is intended to make programs easier to read.
Example from the assignments, the matrix class:

```
matrix m(10,10);  
...  
d = m.get(5,4) - 2*m.get(5,5) + m.get(5,6) ;  
m.set(5,5, d/(dx*dx));
```

With operator overloading we can simplify the syntax:

```
m(5,5) = (m(5,4) - 2*m(5,5) + m(5,6) )/(dx*dx);
```


Overloading operators

- Assume @ to be a placeholder for an operator (e.g. +), then the general syntax for defining this operator within the class is
`return_type operator@(arguments)`
- Unitary operators like +=, -=, ..., ++ usually modify the object that calls them, hence they are implemented as member functions
- Binary operators like +, -, ... need two arguments and typically do not modify the calling object, hence they are often implemented outside the class (potentially being a friend of the class if access to private members is required)
- We will demonstrate the implementation of typical unitary and binary operators for the Vec class shown in previous examples

Overloading operators

- We begin by adding two assignment operators = to the class. They differ in the argument they accept: The first accepts a double value, the second another vector.

```
#include <stdlib.h>
//-----
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v); // Copy constructor
    ~Vec(){ delete[] p;} // Destructor

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
private:
    const int dim;
    double* p;
};
```

```
//-----
void Vec::operator=(const Vec& rhs){
    if(dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] = rhs.p[i];
    else
        exit(1);
}
//-----
void Vec::operator=(const double d){
    for(int i=0; i<dim; i++) p[i] = d;
}
```

Overloading operators

- Next, we add two other unitary operators, += and *= (shown are only the new components of the class)

```
#include <stdlib.h>
//-----
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v);  // Copy constructor
    ~Vec(){ delete[] p;} // Destructor

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
    // Unitary operators
    void operator+=(const Vec& rhs);
    void operator*=(const Vec& rhs);
    void operator*=(const double d);
private:
    const int dim;
    double* p;
};
```

```
void Vec::operator+=(const Vec& rhs){
    if (dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] += rhs.p[i];
    else
        exit(1);
}
//-----
void Vec::operator*=(const Vec& rhs){
    if (dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] *= rhs.p[i];
    else
        exit(1);
}
//-----
void Vec::operator*=(const double d){
    for(int i=0; i<dim; i++) p[i] *= d;
}
```

Overloading operators

- Binary operators as + take two arguments. Consider $c = a+b$; First + is called with a and b as arguments. The operator+ creates a new temporary object t, which is then handed to the assignment operator = as argument.

```
#include <stdlib.h>
//-----
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v); // Copy constructor
    ~Vec(){ delete[] p;} // Destructor

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
    // Unitary operators
    void operator+=(const Vec& rhs);
    void operator*=(const Vec& rhs);
    void operator*=(const double d);
    // Binary operators
    friend Vec operator+(const Vec& a, const Vec& b);
private:
    const int dim;
    double* p;
};
```

```
Vec operator+(const Vec& a, const Vec& b){
    if (a.dim == b.dim){
        Vec t(a);
        t += b;
        return t;
    }
    else
        exit(1);
}
```

Overloading operators

- The subscript operator `[]` takes only one parameter (the index). This is fine for our vector class. For a matrix class however we would like to have the possibility to specify two indices, hence we would overload the function call operator `()` which can take any number of parameters. Both must be member of the class and are typically available in two versions, one `const` (for read access), the other non-`const` (for read/write access).

```
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v); // Copy constructor
    ~Vec(){ delete[] p;} // Destructor

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
    // Unitary operators
    void operator+=(const Vec& rhs);
    void operator*=(const Vec& rhs);
    void operator*=(const double d);
    // index operator
    double operator[](const int idx) const;
    double& operator[](const int idx);
    // Binary operators
    friend Vec operator+(const Vec& a, const Vec& b);
private:
    const int dim;
    double* p;
};
```

```
//-----
double Vec::operator[](const int idx) const{
    return p[idx];
}
//-----
double& Vec::operator[](const int idx){
    return p[idx];
}
```

For completeness, the whole class so far:

```
#include <stdlib.h>
using namespace std;
//-----
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v); // Copy constructor
    ~Vec(){ delete[] p; } // Destructor

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
    // Unitary operators
    void operator+=(const Vec& rhs);
    void operator*=(const Vec& rhs);
    void operator*=(const double d);
    // index operator
    double operator[](const int idx) const;
    double& operator[](const int idx);
    // Binary operators
    friend Vec operator+(const Vec& a, const Vec& b);
private:
    const int dim;
    double* p;
};
//-----
double Vec::operator[](const int idx) const{
    return p[idx];
}
//-----
double& Vec::operator[](const int idx){
    return p[idx];
}
```

```
//-----
Vec operator+(const Vec& a, const Vec& b){
    if (a.dim == b.dim){
        Vec t(a);
        t += b;
        return t;
    }
    else
        exit(1);
}
//-----
void Vec::operator=(const Vec& rhs){
    if(dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] = rhs.p[i];
    else
        exit(1);
}
//-----
void Vec::operator=(const double d){
    for(int i=0; i<dim; i++) p[i] = d;
}
//-----
void Vec::operator+=(const Vec& rhs){
    if (dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] += rhs.p[i];
    else
        exit(1);
}
//-----
void Vec::operator*=(const Vec& rhs){
    if (dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] *= rhs.p[i];
    else
        exit(1);
}
```

```
//-----
void Vec::operator*=(const double d){
    for(int i=0; i<dim; i++) p[i] *= d;
}
//-----
Vec::Vec(const int dim):dim(dim){
    p = new double[dim];
}
//-----
Vec::Vec(const Vec& v):dim(v.dim){
    p = new double[dim];
    for(int i=0; i<dim; i++) p[i] = v.p[i];
}
//-----
int main(){
    Vec v1(10);
    Vec v2(v1);
    Vec v3(10);

    v1 = 1; v2 = 2;
    v1 += v2;
    v3 = v2 + v1;

    v3[9] = 2;
    cout << v3[9] << endl;
}
```

Template functions

- Templates are a very efficient way of writing functions and classes for different parameter types all at once
- For functions that have the same syntax for different datatypes we can generate a universal function accepting (almost) all datatypes by using templates.

```
int max(int i, int j){
    if (i>j)
        return i;
    else
        return j;
}
//-----
double max(double i, double j){
    if (i>j)
        return i;
    else
        return j;
}
//-----
int main(){
    int l = max(3,4);
    double d = max(2.0, 2.5);
}
```

```
template <typename T>
T max(T i, T j){
    if (i>j)
        return i;
    else
        return j;
}
//-----
int main(){
    int l = max(3,4);
    double d = max(2.0, 2.5);
}
```

- Based on the template the compiler will create all versions of the function needed

Template functions

- When we provide not only the general template, but also a version for one particular datatype we call this particular function a *specialization* of the template.

```
#include <iostream>
#include <complex>
using namespace std;
//-----
double maximum(double i, double j){
    cout << "I'm special!" << endl;
    if (i>j) return i;
    else return j;
}
//-----
template <typename T>
T maximum(T i, T j){
    cout << "I'm generic!" << endl;
    if (i>j) return i;
    else return j;
}
//-----
int main(){
    int l = maximum(3,4);
    double d = maximum(2.0, 2.5);

    complex<double> c1(1,-1);
    complex<double> c2(3,2);
    // Problem here: i>j not defined for complex numbers...
    //complex<double> cd = maximum(c1,c2 );
}
```

← Specialization for double. This function could implement totally different things to do for double values.

Output:

I'm generic!
I'm special!

Template classes

- Classes may also have template datatypes in them. Here we will introduce a template version of our vector class, so that we can also have complex values in vectors.

```
template <typename T>
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v); // Copy constructor
    ~Vec(){ delete[] p; } // Destructor

    // Members
    int getDim() const{return dim;}
    void print() const;
    void writeToFile(const std::string fname, const double dx=1,
                    const bool norm=false) const;

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
    // Unitary operators
    void operator+=(const Vec& rhs);
    void operator*=(const Vec& rhs);
    void operator*=(const double d);
    // index operator
    T operator[](const int idx) const;
    T& operator[](const int idx);
    // Binary operators
    friend Vec<T> operator+ <>(const Vec<T>& a, const Vec<T>& b);
private:
    const int dim;
    T* p;
};
```

- When we now want to create a vector object, we also have to specify the datatype of the (numerical) entries:

```
Vec<double> a(10);
Vec< complex<double> > c(10);
```

Template classes

- In our case most of the code is independent of the exact datatype stored in the vectors, so we mainly have to fix the interfaces of the functions:

```
template <typename T>
void Vec<T>::writeToFile(const string fname, const double dx, const bool norm) const{
    ofstream out(fname.c_str());
    if (norm==false)
        for(int i=0; i<dim; i++)
            out << i*dx << "\t" << p[i] << endl;
    else
        for(int i=0; i<dim; i++)
            out << i*dx << "\t" << abs(p[i]) << endl;
    out.close();
}
//-----
template <typename T>
void Vec<T>::print() const{
    for(int i=0; i<dim; i++)
        cout << i << "\t" << p[i] << endl;
}
//-----
template <typename T>
T Vec<T>::operator[](const int idx) const{
    return p[idx-1];
}
//-----
template <typename T>
T& Vec<T>::operator[](const int idx){
    return p[idx-1];
}
```

Template classes

- Before writing the templated version of the vector class, we split the code into header file `vector.hxx`, containing the declarations, and `vector.cxx`, containing the definitions.
- When compiling a program the `vector.cxx` file was compiled into an object and then linked. Everything needed to create the object file was known to the compiler when it read `vector.cxx` and `vector.hxx`.
- Now things are different: We only know the precise datatype for which we have to generate the binary version of the vector class when we compile code that actually uses vector objects. We can not compile any code of the class before we know the datatype. **Hence, we have to include all methods of the class into the header.**
- When you want to use templated friend functions in your class, there are some peculiarities about where to put these functions in the source file. Learn from the templated vector class and from other template classes how to do this.

```
#ifndef VECTOR_HXX_
#define VECTOR_HXX_

#include <string>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <cmath>

//-----
template <typename T> class Vec;
//-----

template <typename T>
Vec<T> operator+(const Vec<T>& a, const Vec<T>& b){
    if (a.dim == b.dim){
        Vec<T> t(a);
        t += b;
        return t;
    }
    else
        exit(1);
}

//-----

template <typename T>
class Vec{
public:
    Vec(const int dim); // Standard constructor
    Vec(const Vec& v); // Copy constructor
    ~Vec(){ delete[] p; } // Destructor

    // Members
    int getDim() const{return dim;}
    void print() const;
    void writeToFile(const std::string fname, const double dx=1,
                    const bool norm=false) const;

    // Assignment operator
    void operator=(const double d);
    void operator=(const Vec& rhs);
    // Unitary operators
    void operator+=(const Vec& rhs);
    void operator*=(const Vec& rhs);
    void operator*=(const double d);
    // index operator
    T operator[](const int idx) const;
    T& operator[](const int idx);
    // Binary operators
    friend Vec<T> operator+ <>(const Vec<T>& a, const Vec<T>& b);

private:
    const int dim;
    T* p;
};
```

```
template <typename T>
void Vec<T>::operator*=(const Vec<T>& rhs){
    if (dim==rhs.dim)
        for(int i=0; i<dim; i++) p[i] *= rhs.p[i];
    else
        exit(1);
}
//-----
template <typename T>
void Vec<T>::operator*=(const double d){
    for(int i=0; i<dim; i++) p[i] *= d;
}
//-----
template <typename T>
Vec<T>::Vec(const int dim):dim(dim){
    p = new T[dim];
}
//-----
template <typename T>
Vec<T>::Vec(const Vec<T>& v):dim(v.dim){
    p = new T[dim];
    for(int i=0; i<dim; i++) p[i] = v.p[i];
}
//-----

#endif /* VECTOR_HXX_ */
```