Object-Oriented Programming for Scientific Computing

Smart Pointers and Constness

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

Summer Semester 2018

C++11 and Dynamic Memory Management

Smart Pointers

C++11 offers a number of so-called smart pointers that can help manage dynamic memory and in particular ensure the correct release of allocated memory.

There are three different types of smart pointers:

- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::weak_ptr<T>`

The template argument `T` specifies the type of object the smart pointer points to. These C++11 smart pointers are defined in the header file `memory`.

Exclusive access: `unique_ptr`

- In the case of `unique_ptrs` there is always *exactly one* smart pointer that owns the allocated data. If this pointer is destroyed (e.g. because the function in which it was defined exits or the destructor of the object to which it belongs is called), then the allocated memory is freed.
- This solves the main issue of normal pointers (raw pointers): keeping allocation and deallocation consistent across a possibly large and complex program.
- Smart pointers and normal pointers (raw pointers) should not be mixed to avoid the risk of unauthorized access to already freed memory or double release of memory. Therefore, the allocation of memory must be placed directly in the constructor call of `unique_ptr`.
- An assignment of a normal pointer to a smart pointer is not possible (but a transfer in the constructor is).

Example for unique_ptr

```
#include <memory>
#include <iostream>

struct foo
{
  void doSomething()
  {}
};

int main()
{
    std::unique_ptr<int> test(new int);
    test = new int;  // not allowed: assignment from raw pointer
    int a;
    test = &a;       // not allowed: assignment from raw pointer
    std::unique_ptr<int> test5(&a); // allowed but dangerous
    *test = 2;       // normal memory access
    std::unique_ptr<int> test2(test.release()); // move to test2
```

Example for unique_ptr

```
    test = std::move(test2); // assignment requires using move
    test.swap(test2);        // exchange with other pointer
    if (test == nullptr)     // comparison
        std::cout << "test is nullptr" << std::endl;
    if (!test2)              // test for existence of object
        std::cout << "test2 is nullptr" << std::endl;
    std::unique_ptr<int[]> test3(new int[32]);  // array
    test3[7] = 12;                              // access to array
    if (test3)                                  // access to raw pointer
        std::cout << "test3 is " << test3.get() << std::endl;
    test3.reset();                              // release of memory
    if (!test3)
        std::cout << "test3 is nullptr" << std::endl;
    std::unique_ptr<foo> test4(new foo); // allocate object
    test4->doSomething();                // use method of object
    std::unique_ptr<FILE, int(*)(FILE*)> filePtr(
     fopen("foo.txt", "w"), fclose);     // Create and close file
}
```

Shared access: `shared_ptr`

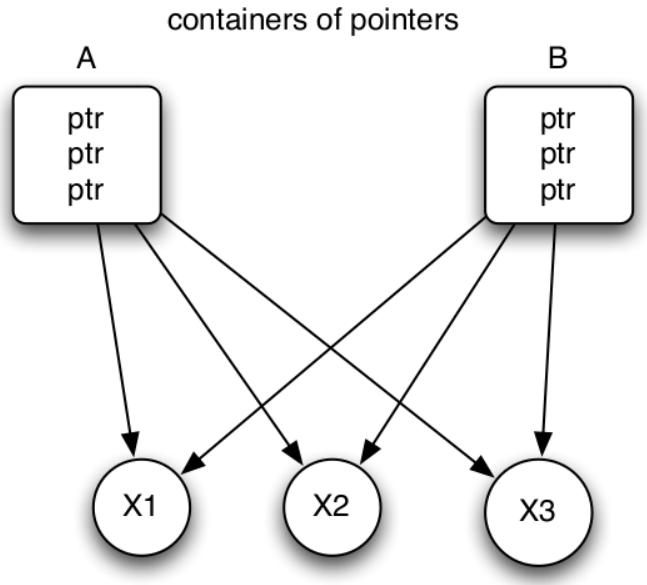

Figure source: D. Kieras, UMichigan

- `shared_ptrs` point to memory that is used concurrently. Several `shared_ptrs` can point to the same memory location.
- The number of simultaneous `shared_ptrs` to the same resource is monitored with reference counting, and the allocated memory is freed when the last `shared_ptr` pointing to it disappears.
- Apart from that, functionality of `shared_ptr` is the same as that of `unique_ptr`.

Implementation of `shared_ptr`

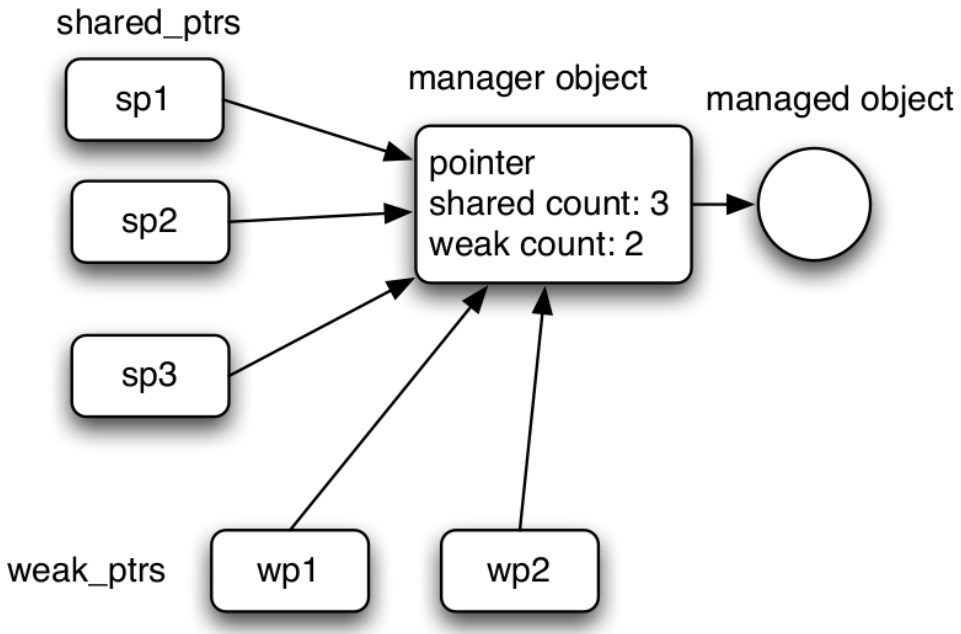

Figure source: D. Kieras, UMichigan

- When the first `shared_ptr` to an object is created, a manager object is created that manages both the allocated resources and a variable that counts how many pointers point to the resources at any given moment.
- For each copy of a `shared_ptr` the counter is incremented, and it is lowered each time a `shared_ptr` is deleted or modified to point to a different location. If the counter reaches zero, the resources are released.

Monitoring resources: `weak_ptr`

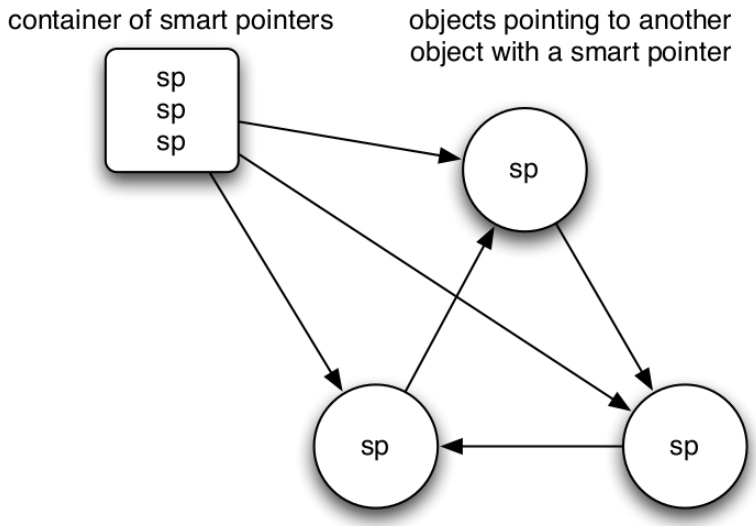

Figure source: D. Kieras, UMichigan

- If several objects have `shared_ptrs` pointing to each other, they can be kept alive artificially after their scope ends, because each object has at least one pointer in the circle pointing to it.
- The class `weak_ptr` has been created to break such a circle (and allow other cases of resource monitoring without ownership).

Monitoring resources: `weak_ptr`

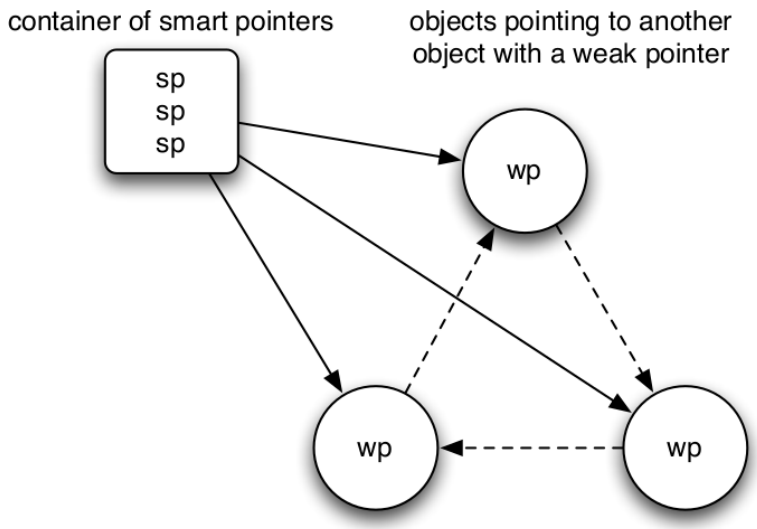

Figure source: D. Kieras, UMichigan

- A `weak_ptr` is not a real pointer. It can not be dereferenced and no methods can be invoked on it.
- A `weak_ptr` only observes a dynamically allocated resource and can be used to check if it still exists.
- If access to the resource is required, the method `lock()` of `weak_ptr` can be used to generate a `shared_ptr` to the resource, ensuring existence of the resource as long it is used.

Shared Pointer Manager Object

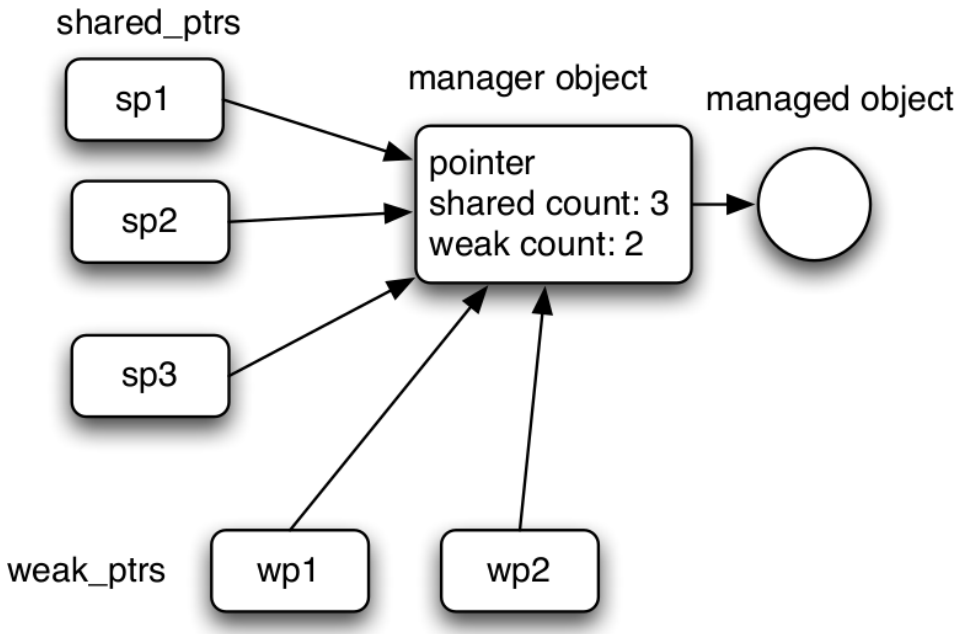

Figure source: D. Kieras, UMichigan

- The manager object of a `shared_ptr` has another counter, the so-called weak counter, which in turn counts the generated `weak_ptrs`. While the allocated resource is released when no `shared_ptr` points to it, the manager object itself is released when in addition no `weak_ptr` points to it.

shared_ptr to this

- Sometimes a pointer for `this` is needed. As one shouldn't mix smart pointers and raw pointers, a `shared_ptr` to `this` must be used.
- If this is realized by `shared_ptr<T> foo(*this)`, then a new manager object will be created and the memory of the object is either not released or released to early.
- Instead, one derives the class from the template class `enable_shared_from_this<T>`. A pointer to `this` is then created with the method `shared_from_this`:

```
shared_ptr<T> foo = shared_from_this();
```

shared_ptr to this

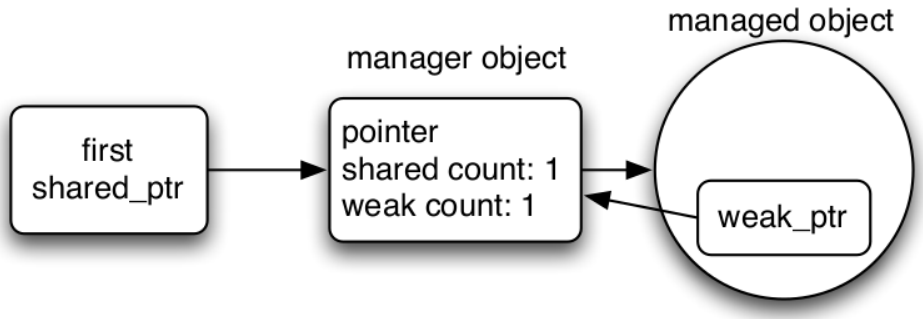

Figure source: D. Kieras, UMichigan

- During the creation of such a derived object in the constructor of a `shared_ptr`, a `weak_ptr` to the object itself is stored within the object.
- This stored `weak_ptr` does not interfere with object destruction, as a stored `shared_ptr` would.
- The method `shared_from_this` simply generates a `shared_ptr` out of the `weak_ptr` and returns it.

Example for shared_ptr

```
#include <memory>
#include <iostream>

class Base : public std::enable_shared_from_this<Base>
{
    void doSomething()
    {
        std::shared_ptr<Base> myObj = shared_from_this();
    }
};

class Derived : public Base
{};

int main()
{
   std::shared_ptr<int> testPtr(new int), testPtr2;
   testPtr2 = testPtr; // increases shared count
   std::cout << testPtr.use_count() << std::endl; // number of shared_ptrs
   testPtr.reset(); // decreases shared count, testPtr is nullptr
```

Example for shared_ptr

```
    // weak pointer example
    std::weak_ptr<int> weakPtr = testPtr2; // increases weak count
    testPtr = weakPtr.lock();
    if (testPtr)
        std::cout << "Object still exists" << std::endl;
    if (weakPtr.expired())
        std::cout << "Object doesn't exist any more" << std::endl;
    std::shared_ptr<int> testPtr3(weakPtr); // throws exception if object has vanished

    // Casting of shared pointers
    std::shared_ptr<Base> basePtr(new Derived);
    std::shared_ptr<Derived> derivedPtr;
    // create cast smart pointer sharing ownership with original pointer
    derivedPtr = std::static_pointer_cast<Derived>(basePtr);
}
```

make_shared and make_unique

The constructor call

```
std::shared_ptr<Base> basePtr(new Derived);
```

can be replaced by the specialized function

```
std::shared_ptr<Base> basePtr = std::make_shared<Derived>();
```

Advantages:

- Only one memory allocation instead of two (object + manager object)
- Raw memory access with `new` is no longer visible

Disadvantage:

- `weak_ptrs` can keep (possibly large) object alive, since manager object is stored in same allocation

Since C++14 there is also `std::make_unique` for consistency, but this behaves just as the constructor call of `unique_ptr`.

Constant Variables

- For constant variables the compiler ensures that the content is not changed during program execution.
- Constant variables must be initialized when they are defined.
- They can't be changed later on.

```
const int numElements = 100;      // initialization
numElements = 200;                // not allowed, const
```

- Compared to macros in C, constant variables are preferred, because they enable strict type checking by the compiler.

Constant References

- References can be also defined as constant. The value pointed to by the reference cannot be changed (using the reference).
- Constant variables only allow constant references (since otherwise they might be changed using the reference).

```
int numNodes = 100;             // variable
const int& nn = numNodes;       // var. can't be changed using nn
                                // but can be using numNodes
const int numElements = 99; // initialization
int& ne = numElements;          // not allowed, const-correctness
                                // wouldn't be guaranteed anymore
const int& numElem = numElements; // allowed
```

- Constant references are a great way to pass a variable to a function without copying:

```
Matrix& operator+=(const Matrix& b);
```

Constant Pointers

For pointers there are two different types of constness. For a pointer it may be forbidden

- to change the contents of the variable to which it points. This is expressed by writing `const` before the type of the pointer:

```
char s[17];
const char* pc = s; // pointer to constant
pc[3] = 'c';        // error, content is const
++pc;               // allowed
```

- to change the address stored in the pointer (such a pointer effectively acts as a reference). This is expressed by writing `const` between the type of the pointer and the name of the pointer:

```
char* const cp = s; // const pointer
cp[3] = 'c';        // allowed
++cp;               // error, pointer is const
```

Constant Pointers

- Of course there is also the combination of both (which corresponds to a constant reference):

```
const char* const cpc = s; // const pointer to constant
cpc[3] = 'c';              // error, content is const
++cpc;                     // error, pointer is const
```

- In general, `const` qualifiers come *after* what they modify, and only refer to what comes after *them* if they are the first word of the type definition:

```
char const * const cpc = s; // same as const char* const
cpc[3] = 'c';               // error, content is const
++cpc;                      // error, pointer is const
```

Constant Objects

- Objects can also be defined as constant.
- The user assumes that the content of a constant object doesn't change. This must be guaranteed by the implementation.
- Therefore, it isn't allowed to call methods that could change the object.
- Functions which will not violate constness are marked by the addition of the keyword `const` after the argument list.
  - The keyword is part of the name. There can be a `const` and a non-`const` variant with the same argument list.
  - Important: the `const` must also be added to the method definition outside of the class.
- For constant objects only `const` methods can be called.

```cpp
#include <iostream>
class X
{
  public:
    int foo() const
    {
        return 3;
    }
    int foo()
    {
        return 2;
    }
};

int main()
{
    X a;
    const X& b = a;
    std::cout << a.foo() << " " << b.foo() << std::endl;
    // produces the output "2 3"
}
```

Example: Matrix Class

```
double* Matrix::operator[](int i)
{
    if (i < 0 || i >= numRows)
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is [0:" << numRows-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return entries[i];
}

const double* Matrix::operator[](int i) const
{
    if (i < 0 || i >= numRows)
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is [0:" << numRows-1 << "]";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return entries[i];
}
```

Using these two definitions we may write:

```
Matrix A(4,6,0.0);
for (int i = 0; i < A.Rows(); ++i)
    A[i][i] = 2.0;
const Matrix E(5,5,1.0);
for (int i = 0; i < E.Rows(); ++i)
    std::cout << E[i][i] << std::endl;
```

Returning a pointer to a constant will prevent the object being implicitly modified by the return value:

```
A[2][3] = -1.0;    // ok, no constant
E[1][1] = 0.0;     // compiler error
```

The returned pointer itself is not constant and can be changed, but this doesn't affect the matrix since it is only a *copy* of `entries[i]`.

Physical and Logical Constness

When is a method `const`?

There are two possible interpretations:

1. The object remains bitwise unchanged. That's how the compiler sees it (that's all it can check) and what it tries to ensure, e.g. by treating all data members of a `const` object also as constants. This is also known as physical constness.
2. The object remains conceptually unchanged for the user of the class. This is referred to as a logical constness. But the compiler is unable to check the semantics.

Physical Constness and Pointers

- In our matrix class example with dynamic memory management, we have used a pointer of type `double**` to store the matrix.
- Making this pointer constant we obtain a pointer of type `double** const`. This way it's only forbidden to change the memory address which is stored in the pointer but not the entries in the matrix.
- The compiler doesn't complain about the definition:

```
double& Matrix::operator()(int i, int j) const;
```

  This therefore allows changing a constant object:

```
const Matrix E(5,5,1.0);
E(1,1) = 0.;
```

- It is even allowed to change the entries within the class itself:

```
double& Matrix::operator()(int i, int j) const
{
    entries[0][0] = 1.;
    return entries[i][j];
}
```

Alternatives

This can be prevented using shared pointers, or simply an STL container as in the first variant of the matrix class:

```
std::vector<std::vector<double>>
```

- In a `const` object this becomes a `const std::vector<std::vector<double>>`.
- Defining the access function

  ```
  double& Matrix::operator()(int i, int j) const;
  ```

  results in an error message from the compiler:

  ```
  matrix.cc: In member function
    'double& Matrix::operator()(int, int) const':
  matrix.cc:63: error: invalid initialization
    of reference of type 'double&'
    from expression of type 'const double'
  ```

Alternatives (II)

- Returning entire row vectors with

  ```
  std::vector<double>& Matrix::operator[](int i) const;
  ```

  fails as well:

  ```
  matrix.cc: In member function
    'std::vector<double, std::allocator<double>>&
    Matrix::operator[](int) const':
  matrix.cc:87: error: invalid initialization
    of reference of type
    'std::vector<double, std::allocator<double>>&'
    from expression of type
    'const std::vector<double, std::allocator<double>>'
  ```

Note: Using pointers it is easy to circumvent the compiler functionality for monitoring physical constness. Therefore it is appropriate to exercise caution when using `const` methods for objects that use dynamically allocated memory. This is not an issue when using smart pointers and STL containers, which manage memory automatically.

Logical Constness and Caches

- Sometimes it is useful to store calculated values with high computational cost in order to save computing time when they are needed several times.
- We add the private variables `double norm` and `bool normIsValid` to the matrix class and make sure that `normIsValid` will always be initialized with `false` in the constructor.
- Then we can implement a method for the infinity norm and cache its result so that we don't need to recompute it if it is required several times.
- Methods that change the matrix can set `normIsValid` to `false` to trigger a recomputation if needed.

Logical Constness and Caches

```
double Matrix::infinityNorm()
{
  if (!normIsValid)
  {
    norm = 0.;
    for (int j = 0; j < numCols; ++j)
    {
      double sum = 0.;
      for (int i = 0; i < numRows; ++i)
        sum += std::abs(entries[i][j]);
      if (sum > norm)
        norm = sum;
    }
    normIsValid = true;
  }
  return norm;
}
```

- This function also makes sense for a constant matrix and doesn't semantically violate constness.
- But the compiler doesn't allow declaring it `const`.

Solution

Solution that allows declaring the method as `const`:

- One defines both variables as `mutable`.

```
mutable bool normIsValid;
mutable double norm;
```

- Members that are `mutable` can also be modified in `const` objects.
- This should only be used when it is absolutely necessary and doesn't change the logical constness of the object.

Possible use cases for `mutable`:

- Caching computed values, as above
- Thread-safe programming for `const` objects
- Temporary debug values in `const` methods

Friend

In some cases it may be necessary for other classes or functions to access the protected members of a class.

Example: Linked list

- `Node` contains the data.
- `List` should be able to change the data of `Node`.
- The data of `Node` should be private.
- `List` is `friend` of `Node` and may therefore access private data.

Friend II

- Classes and free functions can be `friend` of another class.
- Such a `friend` may access the private data of the class.

Example `friend` class:

```
class List;

class Node
{
private:
    Node* next;
public:
    int value;
    friend class List;
};
```

Friend II

- Classes and free functions can be `friend` of another class.
- Such a `friend` may access the private data of the class.

Example `friend` function:

```
class Matrix
{
    friend Matrix invert(const Matrix&);
    // ...
};

...
Matrix A(10);
...
Matrix inv = invert(A);
```

Friend III

- Almost everything that can be written as a class method can also be programmed as a free `friend` function.
- All classes and functions which are `friend` logically belong to the class, as they build on its internal structure.
- Avoid `friend` declarations. They open up encapsulation and raise the cost of maintenance.

Summary

- `unique_ptrs` can be used for dynamic memory management. Only copy constructor and copy-assignment operator require work.
- `shared_ptrs` can be used for transparent sharing of resources. These eliminate almost all work of dynamic memory management.
- `weak_ptrs` can be used to monitor external resources.
- The type of smart pointers makes their purpose explicit: fully owned, shared or monitored resource.
- Correct constness makes compiler optimizations possible and prevents accidental changes of data.
- The keyword `mutable` makes it possible to write logically but not bit-wise constant objects (e.g. caches for efficiency).