Not So Tiny Task №5 (0.5 + 0.5 points)

Implement two variants of ScopedPointer class.

First one with deep copying;
Second one with transferring an ownership;

Follow rules of 3 and 5 where necessary,
but think with your own head!

Not So Tiny Task №6 (2 + 1 points)

Implement a class to work with square matrices.
In this class you should have:

points:

2

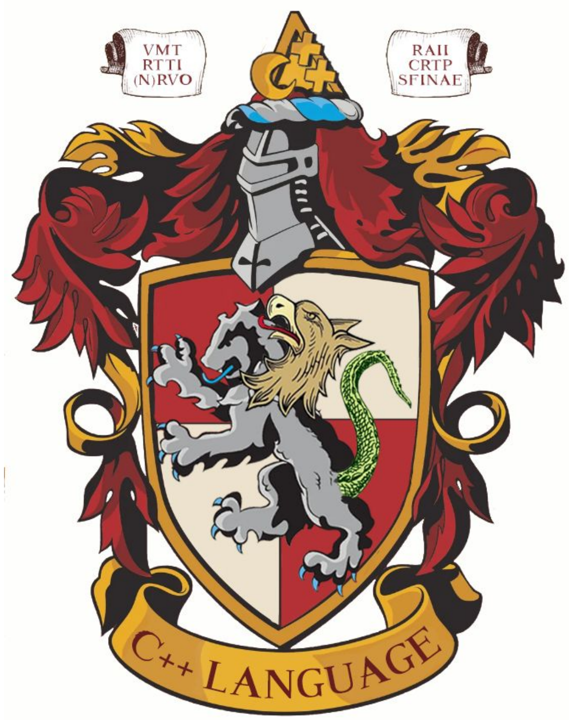
1. Fully correct work with memory (ctr/dstr/assignment etc)
2. Constructor from 1D std::vector of doubles (elements of it will be placed on main diagonal)
3. explicit type conversion operator to double (returns sum of all elements, or det if you really want)
4. Overloaded operators for +, +=, *, *= (for 2 matrices and with scalar), ==, !=

1

5. Double indexing! Matrix m(10); m[5][5] = 42;

System Programming with C++

RAII, casts, operators, friends



RAII

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

RAII

What's wrong with this code?

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

Suppose we indeed do not want to allocate this Triple on the stack on some reason.

RAII

What's wrong with this code?

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

RAII

What's wrong with this code?

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;        copy-paste
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;            copy-paste
   return result;
}
```

RAII

What's wrong with this code?

```
struct Triple { int x; int y; int z; };

int baz() {
    Triple* t = new Triple{13, 42, 1};
    int val =  bar();           What if bar throws an exception? Memory leak!
    if (val > 13) {
        delete t;               copy-paste
        return 0;
    }
    // ... some code that uses t ...
    int result = val + t->x + t->z;
    delete t;                   copy-paste
    return result;
}
```

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

1. We should constantly think about memory for t, don't forget to free it

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

1. We should constantly think about memory for t, don't forget to free it
2. It must be freed on any exit path from the method. Including exceptional one.

RAII

```cpp
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

1. We should constantly think about memory for t, don't forget to free it

2. It must be freed on any exit path from the method. Including exceptional one.

How is it solved in other languages?

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
    Triple* t = new Triple{13, 42, 1};
    int val =  bar();
    if (val > 13) {
        delete t;
        return 0;
    }
    // ... some code that uses t ...
    int result = val + t->x + t->z;
    delete t;
    return result;
}
```

What’s wrong with this code?

1. We should constantly think about memory for t, don’t forget to free it
2. It must be freed on any exit path from the method. Including exceptional one.

How is it solved in other languages?

try/finally, try-with-resources, with

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

1. We should constantly think about memory for t, don't forget to free it
2. It must be freed on any exit path from the method. Including exceptional one.

How to solve this problem here?

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

What's wrong with this code?

1. We should constantly think about memory for t, don't forget to free it
2. It must be freed on any exit path from the method. Including exceptional one.

How to solve this problem here?

constructors and destructors!

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

acquire some resource

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
    Triple* t = new Triple{13, 42, 1};
    int val =  bar();
    if (val > 13) {
        delete t;
        return 0;
    }
    // ... some code that uses t ...
    int result = val + t->x + t->z;
    delete t;
    return result;
}
```

acquire some resource

in our case "resource" is memory, but it can also be: a file, a mutex, a thread, etc.

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};    acquire some resource
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...        use of the resource
   int result = val + t->x + t->z;         use of the resource
   delete t;
   return result;
}
```

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
    Triple* t = new Triple{13, 42, 1};     acquire some resource
    int val =  bar();
    if (val > 13) {
        delete t;                          free the resource
        return 0;
    }
    // ... some code that uses t ...       use of the resource
    int result = val + t->x + t->z;        use of the resource
    delete t;                              free the resource
    return result;
}
```

RAII

```
template<typename T>
class ScopedPointer {
    T* pointer;










};
```

RAII

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }




};
```

RAII

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }


};
```

to make it work with both usual and const ScopedPointers as well as temporary objects

RAII

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

RAII

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }

    T& get() { return *pointer; }
    const T& get() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};
```

the resource which is owned by ScopedPointer

RAII

```
struct Triple { int x; int y; int z; };

int baz() {
   Triple* t = new Triple{13, 42, 1};
   int val =  bar();
   if (val > 13) {
       delete t;
       return 0;
   }
   // ... some code that uses t ...
   int result = val + t->x + t->z;
   delete t;
   return result;
}
```

acquire some resource

free the resource

use of the resource

use of the resource

free the resource

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...

    return val + sp.get().x + sp.get().z;
}
```

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...

    return val + sp.get().x + sp.get().z;
}
```

acquire a resource

Resource **A**cquisition
Is **I**nitialization
(RAII)

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...

    return val + sp.get().x + sp.get().z;
}
```

acquire a resource

From this point, sp owns memory for Triple, it guards it.

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...

    return val + sp.get().x + sp.get().z;
}
```

acquire a resource

From this point, sp owns memory for Triple, it guards it.

When sp is dead, memory is freed.

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};  ← acquire a resource
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...          ← use the resource
                                                  (through  sp)
    return val + sp.get().x + sp.get().z;     ← use the resource
}                                                 (through sp)
```

RAII

We've already seen that with Vector: it owns the memory for the array.

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};   ← acquire a resource
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...          ← use the resource (through  sp)

    return val + sp.get().x + sp.get().z;      ← use the resource (through sp)
}
```

RAII

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

RAII

What's wrong with this class?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

RAII

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }

    T& get() { return *pointer; }
    const T& get() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};
```

What's wrong with this class?

What about rule of 3?
What about rule of 5?

RAII

What's wrong with
this class?

What about rule of 3?
What about rule of 5?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) { ??? }
   ScopedPointer(ScopedPointer&& other) { ??? }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

RAII

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }
    ScopedPointer(const ScopedPointer& other) { ??? }
    ScopedPointer(ScopedPointer&& other) { ??? }

    T& get() { return *pointer; }
    const T& get() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};
```

What's wrong with this class?

What about rule of 3?
What about rule of 5?

What should copy ctr do actually?

RAII

What about rule of 3? What about rule of 5?
What should copy ctr do actually for ScopedPointer?

It depends on what we expect.

RAII

What about rule of 3? What about rule of 5?
What should copy ctr do actually for ScopedPointer?

It depends on what we expect.

First variant: deep copy.

RAII

What about rule of 3? What about rule of 5?
What should copy ctr do actually for ScopedPointer?

It depends on what we expect.

First variant: deep copy. If you pass ScopedPointer by value for example, the content should be fully copied (not only the address, but the content itself!)

RAII

What about rule of 3? What about rule of 5?
What should copy ctr do actually for ScopedPointer?

It depends on what we expect.

First variant: deep copy. If you pass ScopedPointer by value for example, the content should be fully copied (not only the address, but the content itself!)

Second variant: copy constructor should be prohibited at all.

RAII

What about rule of 3? What about rule of 5?
What should copy ctr do actually for ScopedPointer?

It depends on what we expect.

First variant: deep copy. If you pass ScopedPointer by value for example, the content should be fully copied (not only the address, but the content itself!)

Second variant: copy constructor should be prohibited at all. In contrast with move constructor!

RAII

What about rule of 3? What about rule of 5?
What should copy ctr do actually for ScopedPointer?

It depends on what we expect.

First variant: deep copy. If you pass ScopedPointer by value for example, the content should be fully copied (not only the address, but the content itself!)

Second variant: copy constructor should be prohibited at all. In contrast with move constructor! It transfers the ownership.

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};  ← acquire a resource
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    // ... some code that uses sp ...

    return val + sp.get().x + sp.get().z;
}
```

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

   ScopedPointer sp{new Triple{13, 42, 1}};  ← acquire a resource
   int val =  bar();
   if (val > 13) {
       return 0;
   }

   ScopedPointer sp2 = sp;
   // ... some code that uses sp2 ...
   return val + sp2.get().x + sp2.get().z;
}
```

RAII

Let's say the second option was chosen

```
struct Triple { int x; int y; int z; };

int baz() {

   ScopedPointer sp{new Triple{13, 42, 1}};  ← acquire a resource
   int val =  bar();
   if (val > 13) {
       return 0;
   }

   ScopedPointer sp2 = sp; // compilation error here
   // ... some code that uses sp2 ...
   return val + sp2.get().x + sp2.get().z;
}
```

Let's say the second option was chosen

RAII

```
struct Triple { int x; int y; int z; };

int baz() {

    ScopedPointer sp{new Triple{13, 42, 1}};  ← acquire a resource
    int val =  bar();
    if (val > 13) {
        return 0;
    }

    ScopedPointer sp2 = std::move(sp);  ← transfer ownership over
    // ... some code that uses sp2 ...       memory from sp to sp2
    return val + sp2.get().x + sp2.get().z;
}
```

Not So Tiny Task №5 (0.5 + 0.5 points)

Implement two variants of ScopedPointer class.

First one with deep copying;
Second one with transferring an ownership;

Follow rules of 3 and 5 where necessary,
but think with your own head!

RAII

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

In C++ it is quite popular for such classes to mimic (to look like) pointers.

RAII

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + sp.get().x + sp.get().z;
```

← not like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + sp.get().x + sp.get().z;
```

not like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again. dereference operator can be overloaded.

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }

    T& operator*() { return *pointer; }
    const T& operator*() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + (*sp).x + (*sp).z;
```

more like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again. dereference operator can be overloaded. What about selector (arrow)?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& operator*() { return *pointer; }
   const T& operator*() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + (*sp).x + (*sp).z;
```

more like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again. dereference operator can be overloaded. What about selector (arrow)?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& operator*() { return *pointer; }
   const T& operator*() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + sp->x + sp->z;
```

just like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again.
dereference operator can be overloaded.
What about selector (arrow)?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& operator*() { return *pointer; }
   const T& operator*() const { return *pointer; }
   … operator->() { return …; }
   const … operator->() const { return …; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + sp->x + sp->z;
```

just like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again.
dereference operator can be overloaded.
What about selector (arrow)?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& operator*() { return *pointer; }
   const T& operator*() const { return *pointer; }
   T* operator->() { return pointer; }
   const T* operator->() const { return pointer; }
};

ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + sp->x + sp->z;
```

just like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again. dereference operator can be overloaded. What about selector (arrow)?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& operator*() { return *pointer; }
   const T& operator*() const { return *pointer; }
   T* operator->() { return pointer; }
   const T* operator->() const { return pointer; }
};
```

What?? Why?

```
ScopedPointer sp{new Triple{13, 42, 1}};
int result = val + sp->x + sp->z;
```

just like pointer

RAII

In C++ it is quite popular for such classes to mimic (to look like) pointers.

It means that access should also look like access via pointer.

Operators overloading will help again.
dereference operator can be overloaded.
What about selector (arrow)?

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }

   T& operator*() { return *pointer; }
   const T& operator*() const { return *pointer; }
   T* operator->() { return pointer; }
   const T* operator->() const { return pointer; }
};
```

What?? Why?

```
sp->x <=> (sp.operator->())->x
```

this is special semantics for overloaded -> operator. And it is even more interesting!

```cpp
struct A { int x, y; };

struct B {
   A* value;
   A* operator->() const { return value; }
};

struct C {
   B value;
   B operator->() const { return value; }
};

struct D {
   C value;
   C operator->() const { return value; }
};
```

```cpp
struct A { int x, y; };

struct B {
   A* value;
   A* operator->() const { return value; }
};

struct C {
   B value;
   B operator->() const { return value; }
};

struct D {
   C value;
   C operator->() const { return value; }
};

D d{C{B{new A{1, 2}}}};
std::cout << d->x << std::endl; // 1
```

```cpp
struct A { int x, y; };

struct B {
   A* value;
   A* operator->() const { return value; }
};

struct C {
   B value;
   B operator->() const { return value; }
};

struct D {
   C value;
   C operator->() const { return value; }
};

D d{C{B{new A{1, 2}}}};
std::cout << d->x << std::endl; // 1

(((d.operator->()).operator->()).operator->())->x
```

`operator->` is being invoked until the real pointer is returned

```cpp
struct A { int x, y; };

struct B {
   A* value;
   A* operator->() const { return value; }
};

struct C {
   B value;
   B operator->() const { return value; }
};

struct D {
   C value;
   C operator->() const { return value; }
};

D d{C{B{new A{1, 2}}}};
std::cout << d->x << std::endl; // 1

(((d.operator->()).operator->()).operator->())->x
```

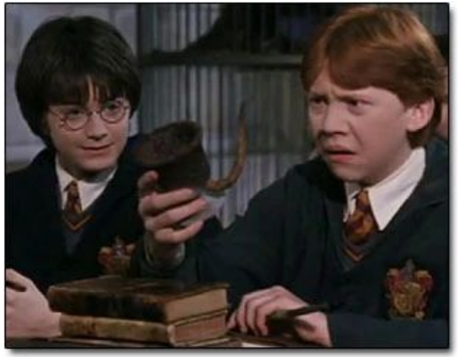
it is called drill down behavior

operator-> is being invoked until the real pointer is returned

Operators overloading

Operators overloading

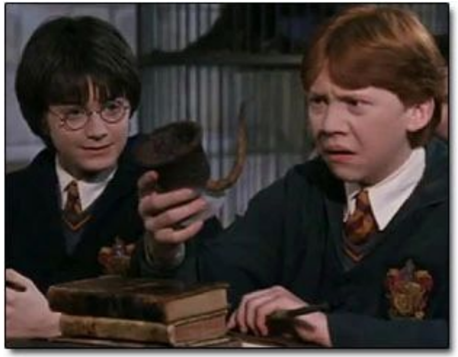
What: in C++ you can "overload" (implement your own) operators for custom classes.



Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

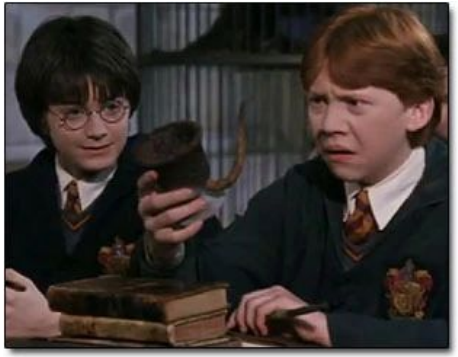
Why?



Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: to make it more convenient to work with them (function call via very laconic operator +).

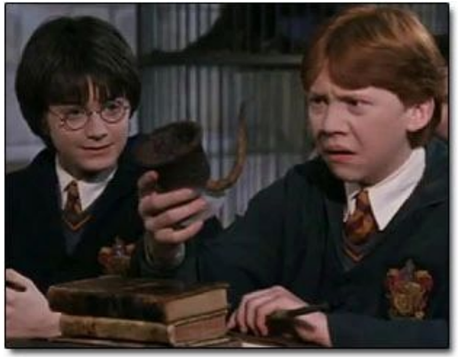


Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: to make it more convenient to work with them (function call via very laconic operator +).

To implement DSLs (Domain Specific Languages)



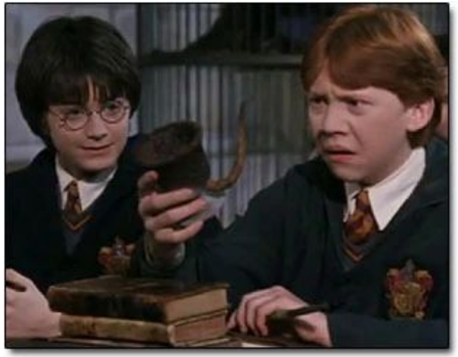
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: to make it more convenient to work with them (function call via very laconic operator +).

To implement DSLs (Domain Specific Languages)

Example: DSL to work with some math objects like matrices in familiar for mathematicians matter (for example A*B gives you matrix multiplication).

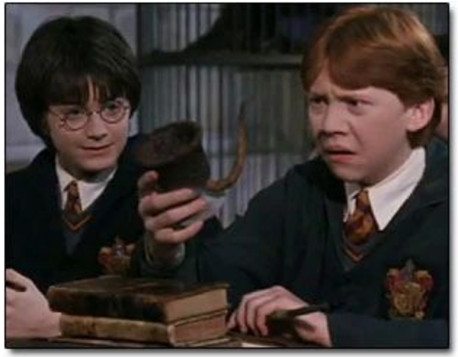


Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload?



Operators overloading

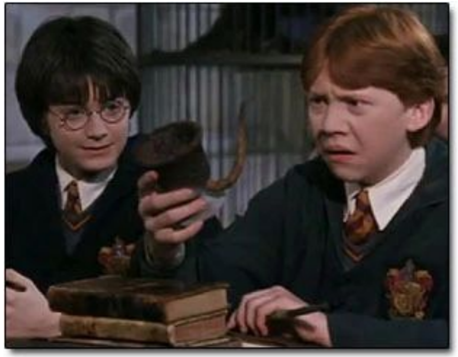
What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload?

Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer



Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

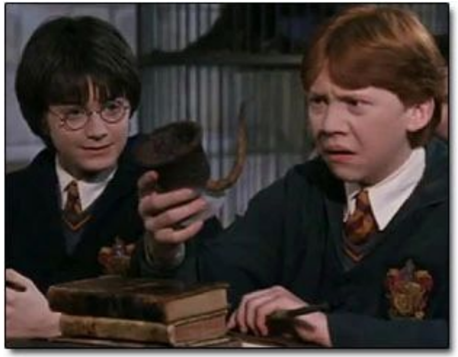
Why: convenience and DSLs.

What can we overload?

Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer

What about casts?



Overloading of type conversions

```
class Date {
   short day_; short month_; short year_;

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

};
```

```cpp
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

};



Date date{"17/3/2024"};
```

```cpp
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

};
```

```cpp
Date date{"17/3/2024"};
std::string str = date;
```

would be nice, this is *implicit* type conversion.

```
class Date {
   short day_; short month_; short year_;

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) {
      // dd/mm/yyyy format string expected
      std::istringstream ss(value); char dummy;
      ss >> day_ >> dummy >> month_ >> dummy >> year_;
   }

   operator std::string() const {
      std::stringstream ss;
      ss << day_ << "/" << month_ << "/" << year_;
      return ss.str();
   }
};

Date date{"17/3/2024"};
std::string str = date;
```

← this will work 😊

```cpp
class Date {
   short day_; short month_; short year_;

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) {
      // dd/mm/yyyy format string expected
      std::istringstream ss(value); char dummy;
      ss >> day_ >> dummy >> month_ >> dummy >> year_;
   }

   explicit operator std::string() const {
      std::stringstream ss;
      ss << day_ << "/" << month_ << "/" << year_;
      return ss.str();
   }
};

Date date{"17/3/2024"};
std::string str = date;
```

compilation error

```cpp
class Date {
   short day_; short month_; short year_;

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) {
      // dd/mm/yyyy format string expected
      std::istringstream ss(value); char dummy;
      ss >> day_ >> dummy >> month_ >> dummy >> year_;
   }

   explicit operator std::string() const {
      std::stringstream ss;
      ss << day_ << "/" << month_ << "/" << year_;
      return ss.str();
   }
};

Date date{"17/3/2024"};
std::string str = (std::string) date;
```

this will work again 😊

```
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

    explicit operator size_t() const {
         return day_ * 1000000 + month_ * 10000 + year_;
    }

};


Date date{"17/3/2024"};
size_t numeric_date = (size_t) date;
```

this will work again 😊

```cpp
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

    explicit operator size_t() const {
        return day_ * 1000000 + month_ * 10000 + year_;
    }

};
```

Here you can specify any type you need. No arguments, no return value.

```cpp
Date date{"17/3/2024"};
size_t numeric_date = (size_t) date;
```

this will work again 😊

Not only C-style casts

Not only C-style casts

What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

Not only C-style casts

```
int x = 42;
double d = (double) x;
```

What do you think about such casts?

Could be implicit, but in other means it is ok! 😊

There are rules how to convert primitive types (since C language), they also work here.

Not only C-style casts

What do you think about such casts?


```
int x = 42;
double d = (double) x;
```

Not only C-style casts

What do you think about such casts?

```
int x = 42;
double d = (double) x;
```


```
const int* cpx = &x;
int* px = (int*) cpx;
```

Not only C-style casts

What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

neutral good

```
const int* cpx = &x;
int* px = (int*) cpx;
```

Quite dangerous already (you've just removed const!),
but predictable and nothing extraordinary bad can happen

Not only C-style casts

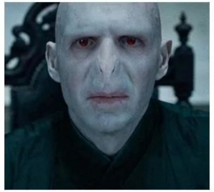
What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

neutral good

```
const int* cpx = &x;
int* px = (int*) cpx;
```

neutral evil



Not only C-style casts

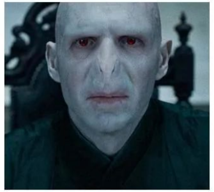
What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

neutral good

```
const int* cpx = &x;
int* px = (int*) cpx;
```

neutral evil



```
size_t spx = (size_t) px;
```

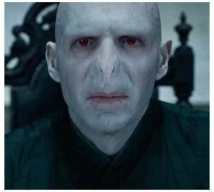
Not only C-style casts

```
int x = 42;
double d = (double) x;
```

What do you think about such casts?

neutral good

neutral evil



```
const int* cpx = &x;
int* px = (int*) cpx;
```

```
size_t spx = (size_t) px;
```

Scary as hell, why to do that?

Not only C-style casts

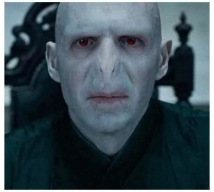
What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

neutral good

```
const int* cpx = &x;
int* px = (int*) cpx;
```

neutral evil



```
float* fpx = (float*) px;
*fpx = 10.0;
```

Scary as hell, why to do that? Direct way to UB

Not only C-style casts

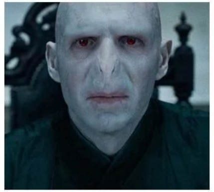
What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

neutral good

```
const int* cpx = &x;
int* px = (int*) cpx;
```

neutral evil



```
size_t spx = (size_t) px;
```

Scary as hell, why to do that? There are some scenarios (write a heap dump to disk?), but it is very dangerous.

Not only C-style casts

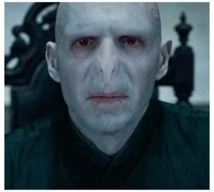
What do you think about such casts?

```
int x = 42;
double d = (double) x;
```

neutral good

```
const int* cpx = &x;
int* px = (int*) cpx;
```

neutral evil



```
size_t spx = (size_t) px;
```

chaotic evil

Not only C-style casts

The problem in C is that all three (quite different) actions can be done with the same language construction: x = (X) y;

```
int x = 42;
double d = (double) x;
```

```
const int* cpx = &x;
int* px = (int*) cpx;
```

```
size_t spx = (size_t) px;
```

Not only C-style casts

The problem in C is that all three (quite different) actions can be done with the same language construction: x = (X) y;

C++ tries to fix that with different types of casts.

```
int x = 42;
double d = (double) x;
```

```
const int* cpx = &x;
int* px = (int*) cpx;
```

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = (int*) cpx;
```

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = (int*) cpx;
```

Compilation errors if used here and here.

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

Cast that is used to remove const/volatile only.

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

Cast that is used to remove const/volatile only.

volatile - modifier that tells the compiler that marked value can be changed from outside of the code.

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

Cast that is used to remove const/volatile only.

volatile - modifier that tells the compiler that marked value can be changed from outside of the code. So, it shouldn't optimize it out.

```
size_t spx = (size_t) px;
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

Cast that is used to remove const/volatile only.

```
size_t spx = reinterpret_cast<size_t>(px);
```

Not only C-style casts

```
int x = 42;
double d = static_cast<double>(x);
```

The safest cast, checks that there are corresponding rules of conversion between types

```
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

Cast that is used to remove const/volatile only.

```
size_t spx = reinterpret_cast<size_t>(px);
```

Still dangerous as hell and can easily call UB, but 1) it is used only for such rude reinterpretation, 2) it is easy to grep it.

```cpp
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

    explicit operator std::string() const {
        std::stringstream ss;
        ss << day_ << "/" << month_ << "/" << year_;
        return ss.str();
    }
};

Date date{"17/3/2024"};
std::string str = (std::string) date;
```

this will work again 😊

```cpp
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

    explicit operator std::string() const {
        std::stringstream ss;
        ss << day_ << "/" << month_ << "/" << year_;
        return ss.str();
    }
};

Date date{"17/3/2024"};
std::string str = static_cast<std::string>(date);
```

this will work again 😊

```cpp
class Date {
   short day_; short month_; short year_;

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) {
      // dd/mm/yyyy format string expected
      std::istringstream ss(value); char dummy;
      ss >> day_ >> dummy >> month_ >> dummy >> year_;
   }

   explicit operator std::string() const {
      std::stringstream ss;
      ss << day_ << "/" << month_ << "/" << year_;
      return ss.str();
   }
};

Date date{"17/3/2024"};
std::string str = static_cast<std::string>(date);
```

1) this is explicit cast, so will work with explicit operator

```cpp
class Date {
    short day_; short month_; short year_;

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) {
        // dd/mm/yyyy format string expected
        std::istringstream ss(value); char dummy;
        ss >> day_ >> dummy >> month_ >> dummy >> year_;
    }

    explicit operator std::string() const {
        std::stringstream ss;
        ss << day_ << "/" << month_ << "/" << year_;
        return ss.str();
    }
};

Date date{"17/3/2024"};
std::string str = static_cast<std::string>(date);
```

1) this is explicit cast, so will work with explicit operator

2) reinterpret_cast will not be compiled here.

Footnotes about casts and type conversions

Footnotes about casts and type conversions

1. `static_cast` will be very useful when we will learn inheritance (soon) as it allows to do a down cast.

Footnotes about casts and type conversions

1. `static_cast` will be very useful when we will learn inheritance (soon) as it allows to do a down cast.

2. `dynamic_cast` is another type of cast, but it is absolutely different. Will discuss it later as well.

Footnotes about casts and type conversions

1. `static_cast` will be very useful when we will learn inheritance (soon) as it allows to do a down cast.

2. `dynamic_cast` is another type of cast, but it is absolutely different. Will discuss it later as well.

3. If you have both type conversion operator and constructor, operator wins.

```
struct Foo {
   Foo() {}
   Foo(const Bar& ref) {
       std::cout << "Foo initialized with Bar in ctor" << std::endl;
   }
};

struct Bar {
   Bar(){}
   operator Foo() {
       std::cout << "Bar to Foo conversion" << std::endl;
       return Foo{};
   }
};


Bar b;
Foo f = b;
```

Two ways to initialize f – copy initialization via ctr or operator Foo() of Bar
What will be printed?

```cpp
struct Foo {
   Foo() {}
   Foo(const Bar& ref) {
       std::cout << "Foo initialized with Bar in ctor" << std::endl;
   }
};

struct Bar {
   Bar(){}
   operator Foo() {
       std::cout << "Bar to Foo conversion" << std::endl;
       return Foo{};
   }
};


Bar b;
Foo f = b;
```

Two ways to initialize f – copy initialization via ctr or operator Foo() of Bar
What will be printed? Operator wins.

```cpp
// Bar to Foo conversion
```

Footnotes about casts and type conversions

1. static_cast will be very useful when we will learn inheritance (soon) as it allows to do a down cast.

2. dynamic_cast is another type of cast, but it is absolutely different. Will discuss it later as well.

3. If you have both type conversion operator and constructor, operator wins.

Footnotes about casts and type conversions

1. static_cast will be very useful when we will learn inheritance (soon) as it allows to do a down cast.

2. dynamic_cast is another type of cast, but it is absolutely different. Will discuss it later as well.

3. If you have both type conversion operator and constructor, operator wins.

4. When compiler decide which type to use, built-in types have priority.

```cpp
struct Bar {
   Bar(){}
   Bar(int v) {}
};

void foo(int) {
   std::cout << "int version of foo is called" << std::endl;
}

void foo(Bar b) {
   std::cout << "Bar version of foo is called" << std::endl;
}


double k = 10.0;
foo(k);
```

```cpp
struct Bar {
   Bar(){}
   Bar(int v) {}
};

void foo(int) {
   std::cout << "int version of foo is called" << std::endl;
}

void foo(Bar b) {
   std::cout << "Bar version of foo is called" << std::endl;
}


double k = 10.0;
foo(k);
```

double is not int neither Bar, but both of them can be casted to int. So, we have a conflict.

```cpp
struct Bar {
   Bar(){}
   Bar(int v) {}
};

void foo(int) {
   std::cout << "int version of foo is called" << std::endl;
}

void foo(Bar b) {
   std::cout << "Bar version of foo is called" << std::endl;
}


double k = 10.0;
foo(k);
```

double is not int neither Bar, but both of them can be casted to int. So, we have a conflict.

In such situations built-in type wins.

```cpp
// int version of foo is called
```

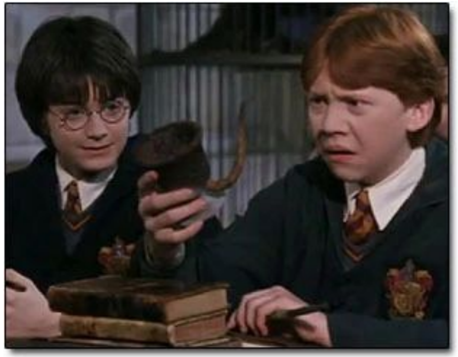
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer
- Type conversions



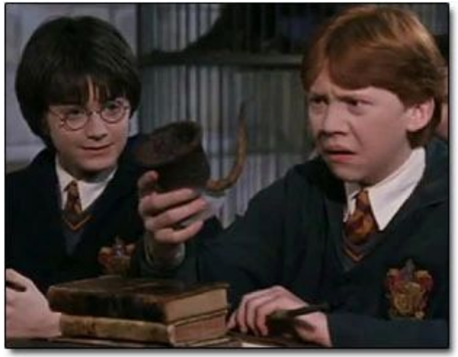
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer
- Type conversions
- Prefix/postfix unary ops?



```cpp
class Date {
   short day_; short month_; short year_;

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) { … }
   operator std::string() const { … }
};


Date date{"17/3/2024"};
```

```cpp
class Date {
   short day_; short month_; short year_;

   void increment() {
      if (day_ < max_day_in_month(month_)) {
         day_++;
      } else {
         day_ = 1;
         ...
      }
   }

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) { … }
   operator std::string() const { … }
};

Date date{"17/3/2024"};
std::string str = date;
```

```
class Date {
   short day_; short month_; short year_;

   void increment() { ... }

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) { … }
   operator std::string() const { … }

   Date& operator++() {
      increment();
      return *this;
   }

};

Date date{"17/3/2024"};
std::string str = date;
```

what is it?

```
class Date {
    short day_; short month_; short year_;

    void increment() { ... }

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) { … }
    operator std::string() const { … }

    Date& operator++() {
        increment();
        return *this;
    }

};

Date date{"17/3/2024"};
++date;
std::cout << (std::string) date; // 18/3/2024
```

what is it? pre-increment!

```cpp
class Date {
    short day_; short month_; short year_;

    void increment() { ... }

public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

    Date(std::string value) { … }
    operator std::string() const { … }

    Date& operator++() {
        increment();
        return *this;
    }

};

Date date{"17/3/2024"};
++date;
std::cout << (std::string) date; // 18/3/2024
```

what is it? pre-increment!

how to add post-increment?

```cpp
class Date {
   short day_; short month_; short year_;

   void increment() { ... }

public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }

   Date(std::string value) { … }
   operator std::string() const { … }

   Date& operator++() { … }
   Date operator++(int) { … }

};

Date date{"17/3/2024"};
++date;
std::cout << (std::string) date; // 18/3/2024
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(std::string value) { … }
   operator std::string() const { … }

   Date& operator++() { … }

   Date operator++(int) {
      Date tmp = *this;
      increment();
      return tmp;
   }
};

Date date{"17/3/2024"};
Date tmp = date++;
std::cout << (std::string) tmp << std::endl;  // ???
std::cout << (std::string) date << std::endl; // ???
```

This is post-increment

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(std::string value) { … }
   operator std::string() const { … }

   Date& operator++() { … }

   Date operator++(int) {
      Date tmp = *this;
      increment();
      return tmp;
   }
};

Date date{"17/3/2024"};
Date tmp = date++;
std::cout << (std::string) tmp << std::endl;  // 17/3/2024
std::cout << (std::string) date << std::endl; // 18/3/2024
```

This is post-increment

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(std::string value) { … }
   operator std::string() const { … }

   Date& operator++() { … }

   Date operator++(int) {
      Date tmp = *this;
      increment();
      return tmp;
   }
};

Date date{"17/3/2024"};
Date tmp = date++;
std::cout << (std::string) tmp << std::endl;  // 17/3/2024
std::cout << (std::string) date << std::endl; // 18/3/2024
```

This is post-increment

That's why many C++ programmers prefer using pre-increment as default variant where possible.

```cpp
for (auto iter = v.begin();
     iter != v.end();
     ++iter) { … }
```

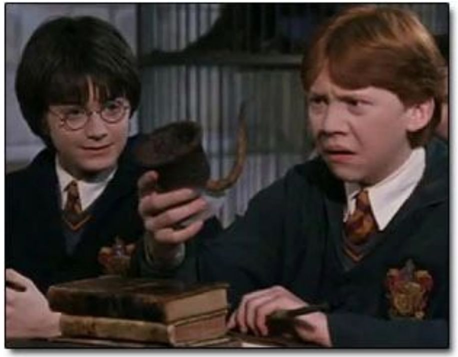
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer
- Type conversions
- Prefix/postfix unary ops
- Any surprises with binary ops?



```
class Date {
    short day_; short month_; short year_;
    void increment() { ... }
public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
    Date(const std::string& value) { … }


    operator std::string() const { … }
};





Date date{"17/3/2024"};
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   // just add one more constructor to make it work with string literals

   operator std::string() const { … }
};



Date date = "17/3/2024";
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       ???
   }

};



Date date = "17/3/2024";
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       ???
   }

};


Date date = "17/3/2024";
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};


Date date = "17/3/2024";
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};


Date date1 = "17/3/2024";
Date date2 = "27/3/2024";
std::cout << date2 - date1 << std::endl; // 10
```

```cpp
class Date {
    short day_; short month_; short year_;
    void increment() { ... }
    int number_of_days() const { ... }
public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
    Date(const std::string& value) { … }
    Date(const char* c_str): Date(std::string{c_str}) { }
    operator std::string() const { … }

    // returns (signed) number of days between dates
    int operator-(const Date& other) {
        return number_of_days() - other.number_of_days();
    }

};


Date date1 = "17/3/2024";
Date date2 = "27/3/2024";
std::cout << date1 - date2 << std::endl; // -10
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};
```

But what about this?

```cpp
Date date = "27/3/2024";
std::cout << date - "17/3/2024" << std::endl;
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};
```

But what about this?
Will it compile?

```cpp
Date date = "27/3/2024";
std::cout << date - "17/3/2024" << std::endl;
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};
```

But what about this?
Will it compile? Yes!

```cpp
Date date = "27/3/2024";
std::cout << date - "17/3/2024" << std::endl;
```

```
class Date {
    short day_; short month_; short year_;
    void increment() { ... }
    int number_of_days() const { ... }
public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
    Date(const std::string& value) { … }
    Date(const char* c_str): Date(std::string{c_str}) { }
    operator std::string() const { … }

    // returns (signed) number of days between dates
    int operator-(const Date& other) {
        return number_of_days() - other.number_of_days();
    }
};
```

Implicitly this ctr will be called

```
Date date = "27/3/2024";
std::cout << date - "17/3/2024" << std::endl; // 10
```

But what about this?
Will it compile? Yes!

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};
```

And what about this?
Will it compile?

```
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};


Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

And what about this?

Will it compile? No! First argument can't be implicitly casted and const char* doesn't know how to substrate a date!

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

   // returns (signed) number of days between dates
   int operator-(const Date& other) {
       return number_of_days() - other.number_of_days();
   }

};


Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

And what about this?
Will it compile? No!

Here where external operator overloading appears.

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

```
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

External operator overloading

```
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

External operator overloading

Do you see any problems here?

```
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

External operator overloading

Do you see any problems here?
`number_of_days` is a private function. Now we can't use it!

```cpp
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
   int number_of_days() const { … }
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

External operator overloading

Do you see any problems here?

`number_of_days` is a private function. Now we can't use it! Move it to public for now.

```
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
   int number_of_days() const { … }
};
```

External operator overloading

```cpp
int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

Do you see any problems here?

`number_of_days` is a private function. Now we can't use it! Move it to public for now.

```cpp
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl; // - 10
```

Now it works 😊

```cpp
class Date {
    short day_; short month_; short year_;
    void increment() { ... }
public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
    Date(const std::string& value) { … }
    Date(const char* c_str): Date(std::string{c_str}) { }
    operator std::string() const { … }
    int number_of_days() const { … }
};

int operator-(const Date& lhs, const Date& rhs) {
    return lhs.number_of_days() - rhs.number_of_days();
}
```

Can we do better than that? We kinda broke the encapsulation 🥺

```cpp
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl; // - 10
```

```
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { … }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }

};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();  // compilation error
}



Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl;
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { … }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
   friend int operator-(const Date&, const Date&);
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}
```

F·R·I·E·N·D·S

```cpp
Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl; // - 10
```

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { … }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(const std::string& value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
   friend int operator-(const Date&, const Date&);
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}


Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl; // - 10
```

F·R·I·E·N·D·S

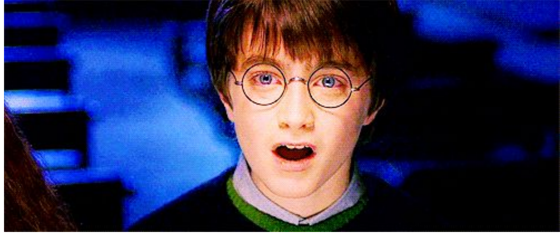
What just happened:

1) We've declared in Date class that we have a friend function operator- and we fully trust it.

```cpp
class Date {
    short day_; short month_; short year_;
    void increment() { ... }
    int number_of_days() const { ... }
public:
    Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
    Date(const std::string& value) { ... }
    Date(const char* c_str): Date(std::string{c_str}) { }
    operator std::string() const { ... }
    friend int operator-(const Date&, const Date&);
};

int operator-(const Date& lhs, const Date& rhs) {
    return lhs.number_of_days() - rhs.number_of_days();
}


Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl; // - 10
```



F·R·I·E·N·D·S

What just happened:

1) We've declared in Date class that we have a friend function operator- and we fully trust it.

2) Now operator- has

Friends: very briefly

`Any class or struct can have some friends.`

Friends: very briefly

Any class or struct can have some friends, including:

1. External functions (they will not have this argument, but will have access to privates)

```cpp
class Date {
   short day_; short month_; short year_;
   void increment() { ... }
   int number_of_days() const { … }
public:
   Date(short d, short m, short y): day_(d), month_(m), year_(y) { }
   Date(std::string value) { … }
   Date(const char* c_str): Date(std::string{c_str}) { }
   operator std::string() const { … }
   friend int operator-(const Date&, const Date&);
};

int operator-(const Date& lhs, const Date& rhs) {
   return lhs.number_of_days() - rhs.number_of_days();
}



Date date = "27/3/2024";
std::cout << "17/3/2024" - date << std::endl; // - 10
```

Friends: very briefly

Any class or struct can have some friends, including:

1. External functions (they will not have this argument, but will have access to privates)

2. Internal functions of other classes (they will have this, but it will be pointer to their host class)

```cpp
class Foo;

class Bar {
   int x;
public:
   void friendlyFunction(Foo* foo);
};
```

```cpp
class Foo;

class Bar {
   int x;
public:
   void friendlyFunction(Foo* foo);
};

class Foo {
   int a, b, c;
   friend void Bar::friendlyFunction(Foo*);
public:
   Foo(int a, int b, int c) : a(a), b(b), c(c) {}
};
```

```cpp
class Foo;

class Bar {
   int x;
public:
   void friendlyFunction(Foo* foo);
};

class Foo {
   int a, b, c;
   friend void Bar::friendlyFunction(Foo*);
public:
   Foo(int a, int b, int c) : a(a), b(b), c(c) {}
};

void Bar::friendlyFunction(Foo *foo) {
   std::cout << foo->a << foo->b << foo->c << this->x;
}
```

Friends: very briefly

Any class or struct can have some friends, including:

1. External functions (they will not have this argument, but will have access to privates)
2. Internal functions of other classes (they will have this, but it will be pointer to their host class)
3. Even other classes! Than all their methods will be friends.

```
class Baz {
   int k, l, m;
   friend class Qux;
public:
   Baz(int k, int l, int m) : k(k), l(l), m(m) {}
};

class Qux {
   Baz body;
public:
   Qux(int k, int l, int m) : body(k, l, m) {}

   void print() {
       std::cout << body.k << body.l << body.m;
   }
};
```

```
class Baz {
    int k, l, m;
    friend class Qux;
public:
    Baz(int k, int l, int m) : k(k), l(l), m(m) {}
};

class Qux {
    Baz body;
public:
    Qux(int k, int l, int m) : body(k, l, m) {}

    void print() {
        std::cout << body.k << body.l << body.m;
    }
};
```

declaration of a friend class Qux

private access to Baz internals

Friends: footnotes

1. Having a friend is not a symmetric relation!

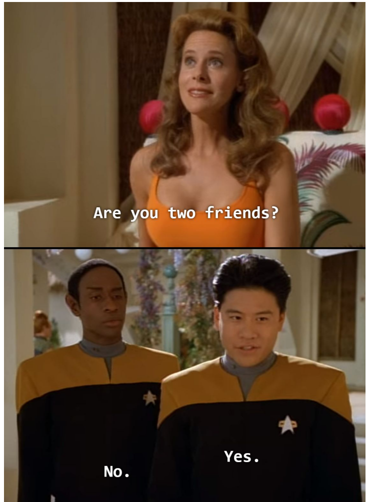
Friends: footnotes

Friends: footnotes

1. Having a friend is not a symmetric relation!

   Declaring someone a friend doesn't mean you will be able to take a look at his internals.

   (just like in real life)



Friends: footnotes

1. Having a friend is not a `symmetric` relation!
2. Your `friend` function will never have you as `this`.

```cpp
class Foo {
    int a, b, c;
    friend void friendlyFunction(Foo*);
public:
    Foo(int a, int b, int c): a(a), b(b), c(c) { }
};

void friendlyFunction(Foo* foo) {
    std::cout << foo->a << foo->b << foo->c;
}
```

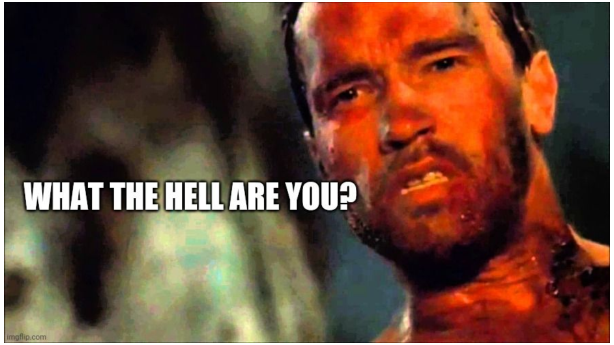
```cpp
class Foo {
    int a, b, c;

    friend void friendlyFunction(Foo* foo) {
        std::cout << foo->a << foo->b << foo->c;
    }
public:
    Foo(int a, int b, int c): a(a), b(b), c(c) { }
};
```

```cpp
class Foo {
    int a, b, c;

    friend void friendlyFunction(Foo* foo) {
        std::cout << foo->a << foo->b << foo->c;
    }
public:
    Foo(int a, int b, int c): a(a), b(b), c(c) { }
};
```

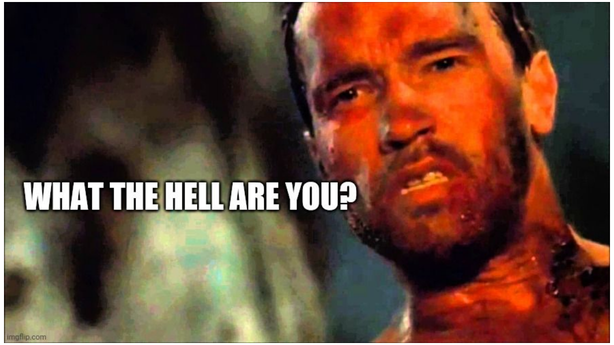
internal function without this!



```cpp
class Foo {
    int a, b, c;

    friend void friendlyFunction(Foo* foo) {
        std::cout << foo->a << foo->b << foo->c << this->a; // compilation
    }                                                       // error
public:
    Foo(int a, int b, int c): a(a), b(b), c(c) { }
};
```

internal function without this!



Friends: footnotes

1. Having a friend is not a symmetric relation!
2. Your friend function will never have you as this.
3. Friends somehow ruin the encapsulation of class.

F·R·I·E·N·D·S

Friends: footnotes

1. Having a friend is not a symmetric relation!

2. Your `friend` function will never have you as `this`.

3. Friends somehow ruin the encapsulation of class.

   (somehow because, well, you've given the access voluntarily, right? You declared your friends)

Friends: footnotes

1. Having a friend is not a symmetric relation!

2. Your friend function will never have you as this.

3. Friends somehow ruin the encapsulation of class.

   (somehow because, well, you've given the access voluntarily, right? You declared your friends)

   And things become even worse with templates! Will discuss it later.

Friends: footnotes

1. Having a friend is not a symmetric relation!

2. Your friend function will never have you as this.

3. Friends somehow ruin the encapsulation of class.

   (somehow because, well, you've given the access voluntarily, right? You declared your friends)

   And things become even worse with templates! Will discuss it later.

4. Nevertheless, it can be useful in some situations.

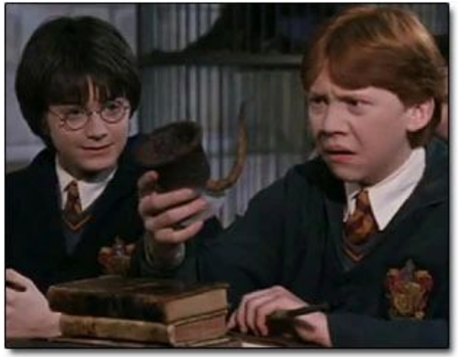
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer
- Type conversions
- Prefix/postfix unary ops
- Binary ops



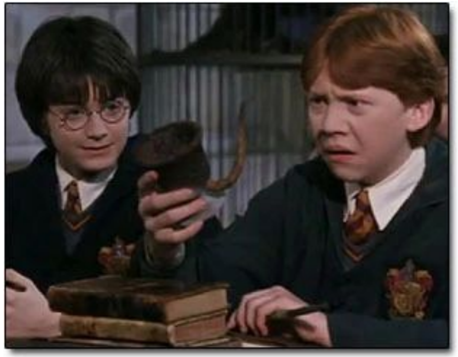
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer
- Type conversions
- Prefix/postfix unary ops
- Binary ops
- Indexing



Indexing

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector(const Vector& other) ... { ... }
    Vector& operator=(const Vector& other) { ... }
    ~Vector() { ... }
    ...
};
```

Indexing

```
Vector v{32};
v.push(13);
v.at(0) = 42;
std::cout << v.at(0);
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector(const Vector& other) ... { ... }
    Vector& operator=(const Vector& other) { ... }
    ~Vector() { ... }

    int& at(size_t index) {
        return data_[index];
    }
};
```

Indexing

```
Vector v{32};
v.push(13);
v[0] = 42;
std::cout << v[0];
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector(const Vector& other) ... { ... }
    Vector& operator=(const Vector& other) { ... }
    ~Vector() { ... }

    int& operator[](size_t index) {
        return data_[index];
    }
};
```

Indexing

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector(const Vector& other) ... { ... }
    Vector& operator=(const Vector& other) { ... }
    ~Vector() { ... }

    int& operator[](size_t index) {
        return data_[index];
    }
};
```

```
Vector v{32};
v.push(13);
v[0] = 42;
std::cout << v[0];
```

Nothing special here, but let's make it more interesting...

Indexing

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector(const Vector& other) ... { ... }
    Vector& operator=(const Vector& other) { ... }
    ~Vector() { ... }

    int& operator[](size_t index) {
        return data_[index];
    }
};
```

```
Vector v{32};
v.push(13);
v[0] = 42;
std::cout << v[0];
```

Nothing special here, but let's make it more interesting...

What if we will have to only a Vector, but 2D array? Like matrix?

Indexing

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector(const Vector& other) ... { ... }
    Vector& operator=(const Vector& other) { ... }
    ~Vector() { ... }

    int& operator[](size_t index) {
        return data_[index];
    }
};
```

```
Vector v{32};
v.push(13);
v[0] = 42;
std::cout << v[0];
```

Nothing special here, but let's make it more interesting...

What if we will have to only a Vector, but 2D array? Like matrix?

Matrix A(20);
A[10][10] = 20;

???

Not So Tiny Task №6 (2 + 1 points)

Implement a class to work with square matrices.
In this class you should have:

1. Fully correct work with memory (ctr/dstr/assignment etc)
2. Constructor from 1D std::vector of doubles (elements of it will be placed on main diagonal)
3. explicit type conversion operator to double (returns sum of all elements, or det if you really want)
4. Overloaded operators for +, +=, *, *= (for 2 matrices and with scalar), ==, !=

Not So Tiny Task №6 (2 + 1 points)

Implement a class to work with square matrices.
In this class you should have:

1. Fully correct work with memory (ctr/dstr/assignment etc)
2. Constructor from 1D std::vector of doubles (elements of it will be placed on main diagonal)
3. explicit type conversion operator to double (returns sum of all elements, or det if you really want)
4. Overloaded operators for +, +=, *, *= (for 2 matrices and with scalar), ==, !=
5. Double indexing! `Matrix m(10); m[5][5] = 42;`

Not So Tiny Task №6 (2 + 1 points)

Implement a class to work with square matrices.
In this class you should have:

points:

2

1. Fully correct work with memory (ctr/dstr/assignment etc)
2. Constructor from 1D std::vector of doubles (elements of it will be placed on main diagonal)
3. explicit type conversion operator to double (returns sum of all elements, or det if you really want)
4. Overloaded operators for +, +=, *, *= (for 2 matrices and with scalar), ==, !=

1

5. Double indexing! Matrix m(10); m[5][5] = 42;

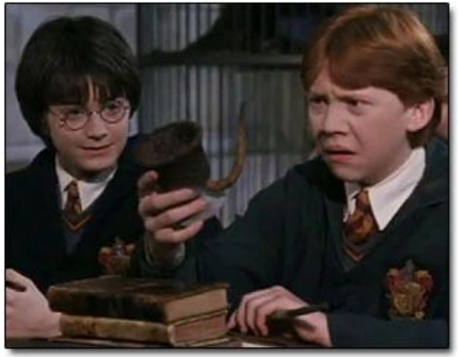
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Already seen:

- Assignment operators,
- + operator for Vector,
- * and -> operators for ScopedPointer
- Type conversions
- Prefix/postfix unary ops
- Binary ops
- Indexing
- …



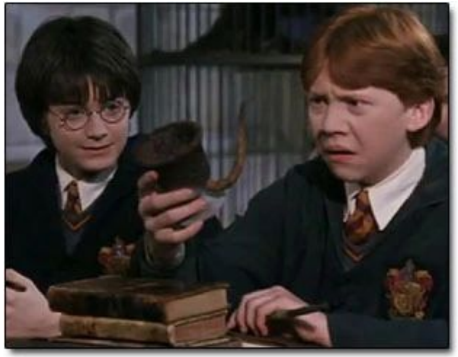
Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++?



Operators overloading

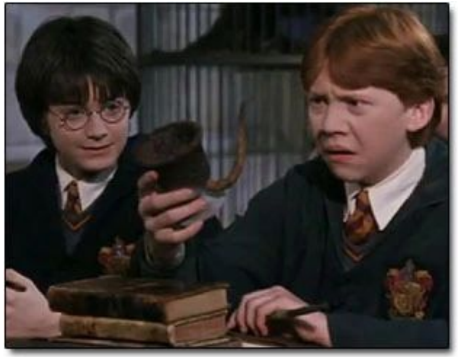
What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++?

1. . (selector operator, like a.b)



Operators overloading

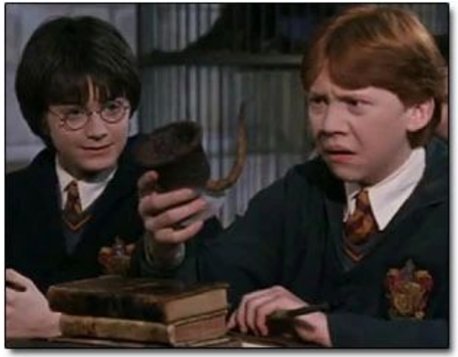
What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++?

1. . (selector operator, like a.b)
2. :: (like Bar::foo)



Operators overloading

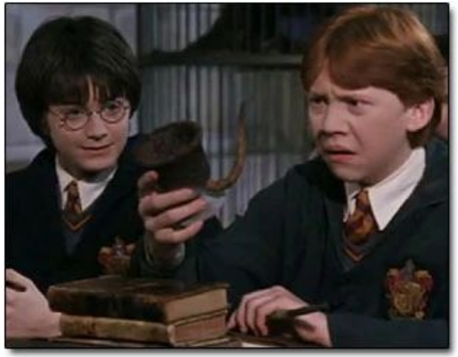
What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++?

1. . (selector operator, like a.b)
2. :: (like Bar::foo)
3. ; (but , can be)



Operators overloading

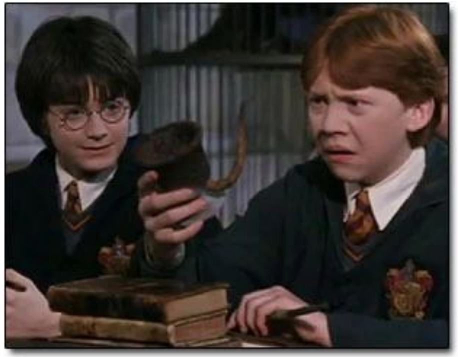
What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++?

1. . (selector operator, like a.b)
2. :: (like Bar::foo)
3. ; (but , can be)
4. ? : (conditional operator)



Operators overloading

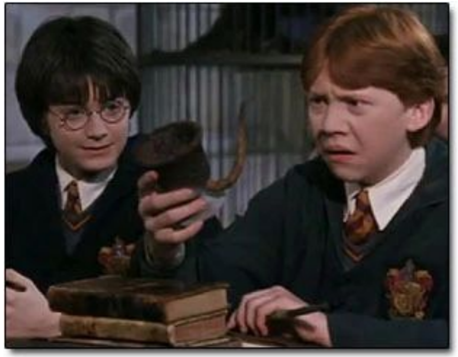
What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can’t we overload in C++?

1. . (selector operator, like a.b)
2. :: (like Bar::foo)
3. ; (but , can be)
4. ? : (conditional operator)
5. sizeof, typeid, static_cast and etc



Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++? Couple of things.

Be careful about:

1. && and || as short-circuit evaluation will be lost

Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++? Couple of things.

Be careful about:

1. && and || as short-circuit evaluation will be lost

   In (a && b) if a evaluates into false, b still can be evaluated in contrast with usual && operator.

Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++? Couple of things.

Be careful about:

1. && and || as short-circuit evaluation will be lost
2. , as sequencing can be lost

Operators overloading

What: in C++ you can "overload" (implement your own) operators for custom classes.

Why: convenience and DSLs.

What can we overload? Well, a lot.

What can't we overload in C++? Couple of things.

Be careful about:

1. && and || as short-circuit evaluation will be lost
2. , as sequencing can be lost
   x = foo(), bar(); no guarantees that foo() evals before bar(). Usually you have it.

Takeaways

- RAII idiom and conception of ownership
- static_cast/const_cast/reinterpret_cast
- Operators overloading general rules and examples
- Friends! Ugly as hell, but can be useful

RAII: Rule of Zero

There is one more "rule" in C++ (along with rule of 3 and 5).

If your class has copy constructor, destructor and etc,

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) { … }
   ScopedPointer(ScopedPointer&& other) { … }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

RAII: Rule of Zero

There is one more "rule" in C++ (along with rule of 3 and 5).

If your class has copy constructor, destructor and etc, than it should deal exclusively with ownership.

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }
    ScopedPointer(const ScopedPointer& other) { … }
    ScopedPointer(ScopedPointer&& other) { … }

    T& get() { return *pointer; }
    const T& get() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};
```

RAII: Rule of Zero

There is one more "rule" in C++ (along with rule of 3 and 5).

If your class has copy constructor, destructor and etc, than it should deal exclusively with ownership. Such classes are called RAII classes.

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }
    ScopedPointer(const ScopedPointer& other) { … }
    ScopedPointer(ScopedPointer&& other) { … }

    T& get() { return *pointer; }
    const T& get() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};
```

RAII: Rule of Zero

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }
    ScopedPointer(const ScopedPointer& other) { … }
    ScopedPointer(ScopedPointer&& other) { … }

    T& get() { return *pointer; }
    const T& get() const { return *pointer; }

    ~ScopedPointer() { delete pointer; }
};
```

There is one more "rule" in C++ (along with rule of 3 and 5).

If your class has copy constructor, destructor and etc, than it should deal exclusively with ownership. Such classes are called RAII classes.

Other classes should have none of copy ctr, dstr, assign operator, move ctr. That's why the rule of 0.

RAII: Rule of Zero

There is one more "rule" in C++ (along with rule of 3 and 5).

If your class has copy constructor, destructor and etc, than it should deal exclusively with ownership. Such classes are called RAII classes.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) { … }
   ScopedPointer(ScopedPointer&& other) { … }

   T& get() { return *pointer; }
   const T& get() const { return *pointer; }

   ~ScopedPointer() { delete pointer; }
};
```

Other classes should have none of copy ctr, dstr, assign operator, move ctr. That's why the rule of 0.

This is the case of **S**ingle **R**esponsibility **R**ule (SRP).