5-spaceship

June 3, 2024

1 The spaceship operator

1.1 Motivation

There are frequent situations where one needs `<` operator for a home-made class. Typically, if you want to make a `std::vector` of such objects, and sort it. But also if you want to use it as a key for e.g. `std::set` or `std::map`.

For completeness, one should also add `>`, `>=`, and `<=`, implemented reusing either `<` and `==`, or `<` and `>`.

Those operators should be defined as free functions, optionally friends, so that left and right arguments will be similarly convertible.

Much boilerplate code to write. Too much.

1.2 Idea

C++20 introduces the *spaceship* operator : `<=>`. Well, the real official name is *three-way comparison operator*.

It is provided by default for all predefined types, and returns *something* which can be compared to `0` (similar to `std::strcmp`), meaning *lower than* if this *something* is lower than `0`, *greater than* if it is greater than `0`, and *equivalent* if it is equal `0`:

[27]:
```
%%file tmp.compare.h

#include <iostream>

template <typename T>
void compare( T lhs, T rhs )
 {
  auto result = (lhs<=>rhs) ;
  if (result<0) std::cout<<lhs<<" < "<<rhs<<std::endl ;
  else if (result>0) std::cout<<lhs<<" > "<<rhs<<std::endl ;
  else if (result==0) std::cout<<lhs<<" ~ "<<rhs<<std::endl ;
  else std::cout<<lhs<<" ? "<<rhs<<std::endl ;
 }
```

```
Overwriting tmp.compare.h
```

[28]:
```
%%file tmp.spaceship.cpp

#include "tmp.compare.h"

int main()
 {
  compare(1,2) ;
  compare(2,2) ;
  compare(2,1) ;
 }
```

```
Overwriting tmp.spaceship.cpp
```

[29]:
```
!rm -f tmp.spaceship.exe && g++ -std=c++20 tmp.spaceship.cpp -o tmp.spaceship.
 ↪exe
```

[30]:
```
!./tmp.spaceship.exe
```

```
1 < 2
2 ~ 2
2 > 1
```

1.3 Different kinds of ordering

The real return type of `<=>` for integers is `std::strong_ordering`: whatever the values, you will always get `true` for exactly one test among `<0`, `==0`, and `>0`.

On the contrary, the return type of `<=>` for floating point numbers is `std::partial_ordering`, because sometimes all tests may return `false`, typically if one number is `NaN`.

[31]:
```
%%file tmp.spaceship.cpp

#include "tmp.compare.h"

int main()
 {
  compare(+0.,-0.) ;
  compare(0./1.,1./0.) ;
  compare(0.,0./0.) ;
 }
```

```
Overwriting tmp.spaceship.cpp
```

[32]:
```
!rm -f tmp.spaceship.exe && g++ -std=c++20 tmp.spaceship.cpp -o tmp.spaceship.
 ↪exe
```

[33]:
```
!./tmp.spaceship.exe
```

```
0 ~ -0
0 < inf
```

```
0 ? -nan
```

Between the two, we have a class `std::weak_ordering`, where `==0` mean that the two compared values are equivalent from a ranking point of view, but not necessarily *equal*: in a given expression, one cannot substitutes one value for the other and be sure to have the same result.

I am not aware of some builtin type whose `<=>` would return an instance of `std::weak_ordering`, but it may make sense for some home-made class, such as the following.

1.4 Home-made class

In the example below, we define a very basic class for positive rational numbers, and provide an implementation of `<=>`:

[34]:
```
%%file tmp.rational.h

#include <iostream>

struct Rational
 {
  unsigned n, d ;

  friend std::weak_ordering operator<=>( Rational const & lhs, Rational const &␣
 ↪rhs )
   { return (lhs.n*rhs.d)<=>(lhs.d*rhs.n) ; }

  friend std::ostream & operator<<( std::ostream & os, Rational const & r )
   { return (os<<r.n<<'/'<<r.d) ; }
} ;
```

```
Writing tmp.rational.h
```

[37]:
```
%%file tmp.spaceship.cpp

#include "tmp.rational.h"
#include "tmp.compare.h"

int main()
 {
  compare<Rational>({ 3, 4 },{ 2, 3 }) ;
  compare<Rational>({ 3, 6 },{ 2, 3 }) ;
  compare<Rational>({ 1, 2 },{ 2, 4 }) ;
 }
```

```
Overwriting tmp.spaceship.cpp
```

[38]:
```
!rm -f tmp.spaceship.exe && g++ -std=c++20 tmp.spaceship.cpp -o tmp.spaceship.
 ↪exe
```

[39]:
```
!./tmp.spaceship.exe
```

```
3/4 > 2/3
3/6 < 2/3
1/2 ~ 2/4
```

Despites `<=>` for `unsigned` returning a `std::strong_ordering`, we prefer here to convert it to `std::weak_ordering`. This way, we emphasize that if `a<=>b` is equal to `0`, it only means that `a` and `b` are logically equivalent, but may lead to different results in other expressions (e.g. printing them).

As one can see from the following code, we have defined only `<=>`, but `<` and `>` (and `<=` and `=>`) work as well. The compiler can rephrase any use of the latter operators in terms of $\{<=>\}$. *Of course, the author of { it Rational} can also provide his own implementations.*

The compiler is NOT able to rephrase `==` and `!=` in terms of `<=>`. Those operators are generally expected to mean equal, *rather than* equivalent. *When `<=>` does not enable a strong order, it is generally advised not to define `==` in terms of `<=>`.*

[27]:
```
%%file tmp.compare.h

#include <iostream>

template <typename T>
void compare( T lhs, T rhs )
 {
  if (lhs<rhs) std::cout<<lhs<<" < "<<rhs<<std::endl ;
  else if (lhs>rhs) std::cout<<lhs<<" > "<<rhs<<std::endl ;
  else if ((lhs<=>rhs)==0) std::cout<<lhs<<" ~ "<<rhs<<std::endl ;
  else std::cout<<lhs<<" ? "<<rhs<<std::endl ;
 }
```

```
Overwriting tmp.compare.h
```

The compiler is NOT able to rephrase `==` and `!=` in terms of `<=>`. Those operators are generally expected to mean equal, *rather than* equivalent. *And if you add some `==` operator to your class, it is generally adviced NOT to implement it in terms of `<=>`, especially if `<=>` does not return an instance of `std::strong_ordering`.*

1.5 Default <=> implementation

One can ask the compiler to provide some default implementation for `<=>` and/or `==`. Logically enough, the default `<=>` will compare the first member variable of the two objects, and goes on to the next member variable as long as the current ones are equivalent.

In the previous example, that would not be relevant, because it would compare the numerators first, and conclude that `3/6` is greater than `2/3`:

[46]:
```
%%file tmp.rational.h

#include <iostream>

struct Rational
```

```
 {
  unsigned n, d ;

  friend auto operator<=>( Rational const &, Rational const & ) = default ;

  friend std::ostream & operator<<( std::ostream & os, Rational const & r )
   { return (os<<r.n<<'/'<<r.d) ; }
} ;
```

```
Overwriting tmp.rational.h
```

[47]:
```
%%file tmp.spaceship.cpp

#include "tmp.rational.h"
#include "tmp.compare.h"

int main()
 {
  Rational r1 { 3, 6 }, r2 { 2, 3 } ;
  compare(r1,r2) ;
 }
```

```
Overwriting tmp.spaceship.cpp
```

[48]:
```
!rm -f tmp.spaceship.exe && g++ -std=c++20 tmp.spaceship.cpp -o tmp.spaceship.
 ↪exe
```

[49]:
```
!./tmp.spaceship.exe
```

```
3/6 > 2/3
```

On the contrary, for some tuple-like class, it may make sense:

[69]:
```
%%file tmp.grade.h

#include <iostream>
#include <string>

struct Grade
 {
  double number ;
  char letter ;
  std::string name ;

  friend auto operator<=>( Grade const &,  Grade const & ) = default ;

  friend std::ostream & operator<<( std::ostream & os, Grade const & g )
   { return (os<<g.letter<<", "<<g.number<<", "<<g.name) ; }
 } ;
```

```
Overwriting tmp.grade.h
```

[70]:
```
%%file tmp.spaceship.cpp

#include "tmp.grade.h"
#include <set>

int main()
 {
  std::set<Grade> grades
   {
     { 19, 'A', "Djamila" },
     { 12, 'C', "Charles" },
     { 16.5, 'A', "Marc" },
   } ;

  for ( auto const & grade : grades )
   { std::cout<<grade<<std::endl ; }
 }
```

```
Overwriting tmp.spaceship.cpp
```

[71]:
```
!rm -f tmp.spaceship.exe && g++ -std=c++20 tmp.spaceship.cpp -o tmp.spaceship.
↪exe
```

[72]:
```
!./tmp.spaceship.exe
```

```
C, 12, Charles
A, 16.5, Marc
A, 19, Djamila
```

We see above that the definition of `<=>` *has been provided, and the use of* `<` *by* `std::set` *to sorts its elements has been rewritten by the compiler in terms of {<=>}.*

Worth to note : if {<=>} is defaulted and no {==} is defined, then the compiler also provides a defaulted {==}. I guess that in most cases, the default implementation of `<=>` *will return some instance of* `std::strong_ordering`*, and it makes sense to also get* `==`*.*

1.6 Summary

- *Defining* `<=>` *allows you to use* `<`*,* `>`*,* `<=`*, and* `>=` *as well.*
- *The standard library defines a few different kinds of order (strong, weak and partial).*
- *Do not confuse equivalence (*`(a<=>b)==0`*) with equality (*`a==b`*).*

2 Questions ?

3 Exercise

We are not fully happy with the default implementation of `<=>`*for our* `Complex` *class below. 1. What happens if I ask the return type of* `<=>` *to be* `std::string_ordering` *instead of* `auto` *? Why*

? 1. Modify it so that the ordering is based on the norm of the complexes. 1. Because you do not use any more the default implementation of `<=>` you had to provide also an implementation for `==`. What happens if you deduce it from `<=>` (using `(((*this)<=>other)==0)`) ? 1. Try to restore the default implementation for `==` only.

[73]:
```
%%file tmp.spaceship.cpp

#include <iostream>
#include <cmath>

struct Complex
 {
  double r, i ;
  double norm() const { return std::sqrt(r*r+i*i) ; }
  friend auto operator<=>( Complex const &, Complex const & ) = default ;
 } ;

std::ostream & operator<<( std::ostream & os, std::partial_ordering cmp )
 { return (os<<'<'<<(cmp<0)<<'|'<<(cmp==0)<<'|'<<(cmp>0)<<'>') ; }

std::ostream & operator<<( std::ostream & os, Complex const & c )
 { return (os<<c.r<<'+'<<c.i<<'i') ; }

template <typename T>
void compare( T lhs, T rhs )
 {
  std::cout<<std::endl ;
  std::cout<<lhs<<" ==  "<<rhs<<" : "<<(lhs==rhs)<<std::endl ;
  std::cout<<lhs<<" <=> "<<rhs<<" : "<<(lhs<=>rhs)<<std::endl ;
 }

int main()
 {
  compare<Complex>({ 1., 2. },{ 1.5, 1.5 }) ;
  compare<Complex>({ 1., 0. },{ 0., 1. }) ;
 }
```

```
Overwriting tmp.spaceship.cpp
```

[74]:
```
!rm -f tmp.spaceship.exe && g++ -std=c++20 tmp.spaceship.cpp -o tmp.spaceship.
 ↪exe
```

[75]:
```
!./tmp.spaceship.exe
```

```
1+2i ==  1.5+1.5i : 0
1+2i <=> 1.5+1.5i : <1|0|0>

1+0i ==  0+1i : 0
```

```
1+0i <=> 0+1i : <0|0|1>
```

4 Sources

- *https://blog.tartanllama.xyz/spaceship-operator/*
- *https://iq.opengenus.org/spaceship-operator-cpp/*
- *https://www.jonathanmueller.dev/talk/meetingcpp2019/*
- *https://quuxplusone.github.io/blog/2021/10/22/hidden-friend-outlives-spaceship/*