4
Abstract through Types

While we have learned to structure behavior into functions, often we have problems where the pre-existing data types are not sufficient to represent a domain concept we are working with. As you already have seen from the standard library C++ allows you to define your own data types in several ways. I am not talking only about the type aliasing we have seen with **`using`** declarations and **`typedef`**, but about defining your own data types together with useful operations. In C++ such user-defined types are first-class citizens and through several language mechanisms they often can be used identical to the built-in types like **`int`**.

4.1 Combining Data and Functions

For example, if you want to represent a calendar date and use that in your program, there is unfortunately no appropriate type in the standard library (yet)[1]. However, in a simple situation a representation of three numbers for the year, the month and the day of the month can be sufficient to represent a person's date of birth.

In C this can be done through a simple struct combining three **`int`** values and that is our starting position for C++ as well. However, we will refine that initial representation significantly throughout this chapter while introducing more and more concepts and syntax.

[1] Before you invent your own, always first look at available library classes. And there is actually a `boost::date` class that provides more functionality we create here.[Gar05]

```
struct Date{
  int year, month, day;
};
```

We can use this date type already, like a corresponding class. As you already learned the keywords **`struct`** and **`class`** are almost equivalent, with the only difference, that the default visibility of members is **`public`** and **`private`** correspondingly. I prefer **`struct`** when all or most members are to be publicly accessible or when it leads to less typing and **`class`** otherwise. However, which keyword to use is often regulated in programming guidelines of a company or project.

Syntax for Classes

A date type that is semantically equivalent to our **struct** Date above can thus also be written like the following code.

```
class Date{
public:
  int year, month, day;
};
```

[2] Please note, that a class definition always ends with a semicolon. Forgetting that semicolon can lead to strange error messages due to the text-inclusion of header files. The resulting syntax error caused by the missing semicolon is often recognized by the compiler in the file where the header was included, or worse, in another header file following the header file with the missing semicolon. The resulting error message will seldom refer to the position of the missing semicolon. Be aware!

[2] Both versions can be used to initialize a Date object like this:

```
Date myBirthday{1964,12,24};
```

However, there is no guarantee that our Date class is actually initialized and also no guarantee that it contains a valid date. For example, if I use the sequence day-month-year when initializing a Date object and then access the individual member variables will result in bogus result, i.e. there is no 1964th of December in year 24. Another problem, that even if the initial date is valid, that there is no protection that any piece of code will not just change one of the **int** values to make the date bogus. These two properties make our current version of Date useless as a parameter type in C++, because no called function can rely on that we actually have a valid date.

Another deficit of our Date type is that we do not yet provide useful functionality for the values created from it. So let us start with a print function to output the date in one common format. The best way to specify its functionality is to actually write a test case for it.

```
void testPrintADate() {
  std::ostringstream os;
  Date day{2012,8,20};
  day.print(os);
  ASSERT_EQUAL("20.08.2012",os.str());
}
```

member function

We implement the printing of the Date type as a const *member function* of our Date class. This brings our class definition into the following state:

```
class Date{
public:
  int year, month, day;
  void print(std::ostream& out) const;
};
```

[3] Covering many different date formats is beyond our scope here.

The format we assume is familiar in central Europe and many parts of the world.[3] We implement our print member function by declaring it in one of the **public** sections of our class Date in its header file Date.h and implement it in the corresponding C++ implementation file Date.cpp as follows.

```
void Date::print(std::ostream& out) const {
  auto ch = out.fill('0');
  out << std::setw(2) << day << '.';
  out << std::setw(2) << month << '.';
  out << std::setw(4) << year;
  out.fill(ch);
}
```

Please note, that we need to remember the original fill character of the stream object and reset it in the end, otherwise the stream's behavior might change after printing a date, e.g., when one also would like to add something else on its output. We better also provide a test case for such behavior to ensure no one is "optimizing away" that behavior.

```
void testPrintADateDoesntChangeFillChar() {
  std::ostringstream os;
  Date day{2012,8,20};
  auto ch=os.fill();
  day.print(os);
  ASSERT_EQUAL(ch,os.fill());
}
```

Let's say, that is enough for printing a date, however, after we have created our first member function for our type, we still face the problem that it is too easy to create invalid dates. We therefore need a means to check, if the provided parameters to form a `Date` actually create a valid date. That might be a bit more complicated than it seams, so we limit ourselves to the Gregorian calendar in those years where it seems to be reasonably valid. I chose the year 1813, since then it was accepted in the last parts in Switzerland where I live[Wik12]. It is hard to set a useful upper limit, since we can not know when there will be a future calendar reform, so I chose the year 9999, because that is the maximum our print format will output nicely.[4]

[4] This will allow future consulting companies make a fortune when we hit the Y10k problem then :-).

The month value must be between 1 for January and 12 for December. However, checking the day in the month is a bit tricky. For all months, except February the number of days per that month are fix, but the exception will be February in leap years. So let us start with a test that checks for `isValidYear(int year)`. We put that function as a member within class `Date`. Because it does not depend on the member variables of a date value, we define it as a *static member function*. In other languages such a function that relates only to a class and not its instances is also called a "class method". In C++ we call it by using the scope-resolution operator, *i.e.*, `Date::isValidYear`. Please note, that for space reasons the code combines tests for within and just out of a boundary in a single function. In more complex environment you should limit your tests to just a single piece of logic within a test.

static member function

```
void testIsValidYearLowerBoundary(){
  ASSERT(Date::isValidYear(1813));
  ASSERT(! Date::isValidYear(1812));
}
void testIsValidYearUpperBoundary(){
  ASSERT(Date::isValidYear(9999));
  ASSERT(! Date::isValidYear(10000));
}
```

The implementation of `Date::isValidYear` then looks like the following code.

```
bool Date::isValidYear(int year) {
  return year >= 1813 && year < 10000;
}
```

Its corresponding declaration as a static member function extends class `Date` like this. Please note, there is no need and also no opportunity to mark a static member function as **`const`**, since it doesn't carry the implicit parameter given by the left-hand side of the `.` operator of a regular member function, *i.e.*, `day` in `day.print(std::cout)`.

```
class Date{
public:
  int year, month, day;
  void print(std::ostream& out) const;
  static bool isValidYear(int year);
};
```

Now, after having limited our range of years, we further need to accommodate a check for leap year condition to be able to implement our check for the days of month within the month of February. Since we have seen the steps of development of a similar static member function already, let us just look at the result.

```
void testIsLeapYear(){
  ASSERT(Date::isLeapYear(1964));
  ASSERT(Date::isLeapYear(2000));
  ASSERT(Date::isLeapYear(2400));
  ASSERT(Date::isLeapYear(2012));
  ASSERT(! Date::isLeapYear(1900));
  ASSERT(! Date::isLeapYear(2013));
  ASSERT(! Date::isLeapYear(2100));
}
```

```
class Date{
//...
  static bool isLeapYear(int year);
};
```

```
bool Date::isLeapYear(int year) {
  return !(year % 4) && ((year % 100) || !(year % 400));
}
```

We can continue the same for a function `isValidMonth(int month)` which is not strictly necessary and can combine all this with a final check for `isValidDate(int year, int month, int date)` as follows.

```
bool Date::isValidDate(int year, int month, int day) {
  if (! isValidYear(year) || day <=0 ) return false;
  switch (month){
  case 1: case 3: case 5: case 7: case 8: case 10:
  case 12: return day <= 31;
  case 4: case 6: case 9: case 11: return day <= 30;
  case 2: return day <=(isLeapYear(year)?29:28);
  }
  return false;
}
```

There are many possibilities to implement the check, such as a lookup table between months and allowed days, but I have opted for a **`switch`** statement to expose you to its syntax. We can combine several cases, because a C++ switch statement is doing a "fall through" to the next case if it doesn't have either a **`break`** or as in our case a **`return`** statement.

4.2 *Initializing Objects and Establishing Invariants*

Now, after we are able to check if a date is actually valid, we need to decide how to handle the erroneous situation, when a user of our class `Date` provides an invalid initialization. As we have learned in section 3.4 there exist several options. But first, we need to see how to create a *constructor* function for our class that initializes our member variables. A constructor of a class is a member function with a slightly different syntax. Its name is identical to the class' name and it doesn't specify a return type, because it actually creates an instance of our class. A constructor, if defined, is the function called when you initialize a variable or a value with the type of your class. The constructor's parameters or other data can be used to initialize the non-static member variables of the class. A constructor for our Date class that will work like the non-constructor (trivial) case we've seen so far looks like the following piece of code if implemented within the class.

constructor

```
class Date{
public:
  int year, month, day;
//...
  Date(int year, int month, int day)
  :year{year},month{month},day{day}{
    // should check here.
  }
};
```

As you can see the signature of the constructor the parameter list is the same like that of a regular function, however, before the function body starts there is an interesting part, beginning with a colon `:`. This part lists the (non-static) member variables to be initialized and provides an initializer for each. The member variables must be given in the same sequence as they are defined within the class. If you omit one of the member variables, it will be initialized with its default value, i.e., zero for **`int`**. A reference member variable must always be initialized that way.[5]

[5] If you use the identical names for the parameters and the member variables of the class. However, when you put code in the constructor body, you have to use special syntax `this->year` to actually access the member variable instead of the parameter `year`.

In the above preliminary version of our class, the constructor function body is empty, but that is where we need to check and establish the invariant that the date is actually valid. But before we do that, we have to decide how to handle the error case. From our five possibilities in section 3.4 we have already out-ruled ignoring the error.

Since we cannot return a specific code from our constructor function, the options "standard result" and "error value" are similar: we would need to have to chose a specific date to mark an invalid input. Selecting such a date would be very arbitrary and if it is outside our defined valid range, it might lead to errors in the code using such a date object. The option to provide the error status through a side effect, as is the case with input formatting errors on a `std::istream` would require an additional reference parameter to our constructor that is also a burden for the users of our class.

This leaves only the remaining option throwing an exception. A very sophisticated implementation might use a specific exception to show what was wrong with the input parameters, but for now, we just select `std::out_of_range` from `<stdexcept>` to denote a problem with the constructor arguments, if they would lead to an invalid date. The exception error reporting allows us to guarantee that only valid dates are represented by our `Date` class. To test that behavior we need to write a test that checks if constructing an invalid date actually throws that exception.[6]

[6] You might wonder, why the attempted creation of the temporary date objects are put within an additional pair of parentheses. `ASSERT_THROWS` is a preprocessor macro and the text-replacement mechanism of the preprocessor would interpret all commas as separating macro parameters resulting in a syntax error. Since the preprocessor would only interpret round parentheses and not curly braces, we need to add the former around our expression that is intended to throw.

```
void testDateCtorThrowsIfInvalid(){
  ASSERT_THROWS((Date{2012,6,31}),std::out_of_range);
  ASSERT_THROWS((Date{0,0,0}),std::out_of_range);
  ASSERT_THROWS((Date{1900,2,29}),std::out_of_range);
  ASSERT_THROWS((Date{2011,0,31}),std::out_of_range);
}
```

Extending our class with the constructor to initialize its member variables allows us to actually hide its internal structure and this ensures that the invariant we ensure is not violated by directly modifying a member variable. This results in the following version of the class definition.

```
class Date{
  int year, month, day;
public:
  void print(std::ostream& out) const;
  static bool isValidYear(int year);
  static bool isLeapYear(int year);
  static bool isValidDate(int year, int month, int day);
  Date(int year, int month, int day);
};
```

The constructor's implementation looks like the following.

```
Date::Date(int year, int month, int day)
:year{year},month{month},day{day}{
  if (! isValidDate(year,month,day))
    throw std::out_of_range{"invalid date"};
}
```

Classes as Abstract Data Types

The version of `class Date` with the now unaccessible private member variables up front still shows that part before the useful parts of the class itself. Beside other reordering that might make it nicer, we can put the useful member functions up front and put the private parts towards the end. To save unnecessary code, we can switch back to using struct for introducing the class type:

```
struct Date{
  void print(std::ostream& out) const;
  static bool isValidYear(int year);
  static bool isLeapYear(int year);
  static bool isValidDate(int year, int month, int day);
  Date(int year, int month, int day);
private:
  int year, month, day;
};
```

This sequencing allows class users to ignore the internal structure more easily and concentrate on the functionality the class provides. Most of your class designs should actually focus on the external behavior and not on its internal representation. For example, this allows the class implementor to vary the implementation, *e.g.*, optimizing it, without affecting the code using the class.[7] It also provides more stable unit tests, since those test will only test the client-side visible public behavior of your class. Accessing private member vari-

[7] Unfortunately C++ requires to recompile the client code, when a class definition is changed. There are means to alleviate that problem such as the PIMPL idiom[Sut11], but we are not caring for that now.

ables in a unit test might seem appropriate, but it closely couples a test to a class' internals, that might change in the future. It is better to check what a client of your class can see of the class' behavior.

abstract data type

A class that is written in such a way that only the behavior given by functions using it and no concrete internal data members are exposed can be viewed as an implementation of an *abstract data type*. This is in contrast to a concrete type like our very first version of `Date` where the only means of using the type is directly accessing its data members.

So far our abstract data type `Date` is not yet very interesting, we only can create valid dates and output them. So let us start with a useful functionality: Why not let the date calculate its following date? However, a unit test for that would require us to be able to check if the date is actually the following date. For the time being, we cheat in that we create two dates and compare if they output the same characters when printed. Because we require that cheating functionality for our tests, we implement that within our unit testing source code and not within the class itself.[8]

[8] Later we will see, how to implement a comparison operator for our class so we can store it in a sorted associative container.

```
bool datesAreEqual(Date const &l, Date const &r){
  std::ostringstream o1;
  l.print(o1);
  std::ostringstream o2;
  r.print(o2);
  return o1.str() == o2.str();
}
void testNextDaySimple(){
  Date aday{2012,8,20};
  aday.nextDay();
  ASSERT(datesAreEqual(aday,Date{2012,8,21}));
}
```

That simple test would allow an implementation that just increments the `day` member variable. But you already know that this might lead to an invalid date, if we reach the end of the current month. So let us test a few other test cases.

```
void testNextDayEndOfMonth(){
  Date eom{2012,8,31};
  eom.nextDay();
  ASSERT(datesAreEqual(eom,Date{2012,9,1}));
}
void testNextDayEndOfShortMonth(){
  Date eosep{2012,9,30};
  eosep.nextDay();
  ASSERT(datesAreEqual(eosep,Date{2012,10,1}));
}
void testNextDayEndOfFeburary(){
  Date eofeb{2012,2,29};
  eofeb.nextDay();
  ASSERT(datesAreEqual(eofeb,Date{2012,3,1}));
}
void testNextDayEndOfFeburaryNonLeapYear(){
  Date eofeb{2013,2,28};
  eofeb.nextDay();
  ASSERT(datesAreEqual(eofeb,Date{2013,3,1}));
}
void testNextDayNotEndOfFeburaryInLeapYear(){
  Date aday{2012,2,28};
  aday.nextDay();
  ASSERT(datesAreEqual(aday,Date{2012,2,29}));
}
```

To implement these tests, we need a function to calculate the last day of the month to effectively compare the day with the last day of the month. When you look closely, you see, that we already have similar functionality hiding in our `isValidDate` static member function. Therefore, we refactor that function, so that we can re-use that functionality in `nextDate`.

```
bool Date::isValidDate(int year, int month, int day) {
  return isValidYear(year) && day > 0 &&
      day <= endOfMonth(year,month);
}
int  Date::endOfMonth(int year, int month){
  switch (month){
  case 1: case 3: case 5: case 7: case 8: case 10:
  case 12: return 31;
  case 4: case 6: case 9: case 11: return  30;
  case 2: return (isLeapYear(year)?29:28);
  }
  return 0;
}
```

Now, we are ready to implement `nextDay` as follows.

```
void Date::nextDay() {
  if (day == endOfMonth(year,month)){
    day = 1;
    if (month == 12){
      month=1;
      ++year;
      if (!isValidYear(year))
        throw std::out_of_range("last year");
    } else {
      ++month;
    }
  } else {
    ++day;
  }
}
```

Note that `nextDay` is quite complex and that we actually didn't test the year change yet. The code might be simplified by extracting calculating the next month/year combination, if necessary, but for the moment let's stick to it. Here are some tests for the year ending to ensure our code actually works.

```
void testNextDayEndOfYear(){
  Date silvester{2012,12,31};
  silvester.nextDay();
  ASSERT(datesAreEqual(silvester,Date{2013,1,1}));
}
void testNextDayEndOfLastAllowedYear(){
  Date lastdayever{9999,12,31};
  ASSERT_THROWS(lastdayever.nextDay(),std::out_of_range);
}
```

4.3 *Making a Class Sortable - Overlaod Relational Operators*

To actually sort values of a type, like our `Date` class requires us to implement the less-than comparison operator. This is done by implementing a function named **`operator<`** that takes 2 arguments and returns a **`bool`**. We usually define the parameters to be passed by const-reference, because a relational operator like < shouldn't change its arguments.[9]

[9] You already have seen a glimpse of operator overloading with the operator function names in Table 1.3 on page 40.

```
bool operator<(Date const&, Date const&);
```

Because comparing a date requires us to look at its internal data, we need to implement it as a member function of `Date`. Each member function takes the **`this`** object as its implicit first parameter, so the member function **`operator<`** only requires one parameter to get the second operand of our **`operator<`**. And because also the first parameter should be passed by const-reference we define the **`operator<`** as a const member function.

```
bool Date::operator<(Date const &)const{ /*. . . */}
```

Implementing a First Overloaded Operator Function

Before we implement that, it is better to define some test cases to ensure we are doing it right. First, we test if two adjacent dates compare correct. Please note, that in contrast to built-in operators, we have to ensure that exchanging the operands returns **false**, because a user-implemented overloaded operator can have any semantic, but better follows the intuitive rules.[10] We also check that a day is not less than itself, because otherwise our **operator<** would not fulfill the requirements for our sorted associative containers.

[10] "When in doubt, do as the ints do." Scott Meyers.[Mey05]

```
void testDateLessThanAdjacentDates(){
  Date aday{2012,8,13};
  Date later{2012,8,14};
  ASSERT(aday < later);
  ASSERT(!(later < aday)); // asymmetric
  ASSERT(!(aday < aday)); // not reflexive
}
```

Now we can start to implement our operator. We extend the class definition with its declaration. The parameter name `rhs` stands for the "right-hand side" operand, whereas the left operand is implicitly given by the **this** object.

```
class Date{
//...
  bool operator <(Date const& rhs) const;
};
```

And add the corresponding implementation.

```
bool Date::operator <(Date const& rhs) const {
  return year < rhs.year ||
    (year == rhs.year && (month < rhs.month ||
      (month==rhs.month && day < rhs.day)));
}
```

As you might recognized the implementation does more than what we tested so far. Therefore, we add additional tests for dates that have a different year and month. This should ensure that each relation in the complex expression of the implementation is checked at least once both ways.

```
void testDateLessThanYearOff(){
  Date aday{2011,8,13};
  Date yearlater{2012,8,13};
  ASSERT(aday < yearlater);
  ASSERT(!(yearlater < aday));
}
```

```
void testDateLessThanMonthOff(){
  Date aday{2012,7,13};
  Date later{2012,8,13};
  ASSERT(aday < later);
  ASSERT(!(later < aday));
}
```

If you have other classes where it is easy to define comparison by comparing the member variables with a given priority, *e.g.*, first sort by year then by month and last by day we can use the comparison operators from `std::tuple` to implement comparison for our type, avoiding boilerplate code. The less-than **`operator<`** can be implemented by using the tuple factory function `std::tie` that does not copy the values into the tuple, but creates a tuple of (**`const`**-) references as follows if the header `<tuple>` is included first.

```
bool Date::operator <(Date const& rhs) const {
    return std::tie(year,month,day) <
        std::tie(rhs.year,rhs.month,rhs.day);
}
```

Providing a Complete Set of Relational Operators

After we have implemented the less-than comparison with **`operator<`** that is required by `std::set<Date>`, for example, the other relational operators can also be useful for a programmer. Instead of copying the implementation of **`operator<`** and adjusting its logic, it is much easier to rely on **`operator<`** and use it directly to implement the other relational operators. For example, to test if `a > b`, we can use `b < a` and thus **`operator<`**, or `a >= b` can be translated to `!(a < b)`. With this idea we can implement the rest of a full set of relational operators without accessing the class' internals and thus have less coupling in the code and fewer places to change if we need to change the internal representation. Therefore, we put the operator functions outside of the class. In addition it is wise to implement these as **`inline`** functions directly in the header of the class, because they are so simple that a direct inlining of their code by a compiler can be more optimal than a function call instead and we do not have the burden to navigate from declaration to definition and back.

inline function

A few test cases do not harm us, even so the implementations are trivial. But those help us to avoid typos. Please note, that for the "or equal" cases we check, that those operators are reflexive, *i.e.*, they return true when a value is compared with itself.

```
void testDateGreaterThan(){
  Date aday{2012,2,29};
  Date later{2012,3,1};
  ASSERT(later > aday);
  ASSERT(!(aday > later));
}
```

```
void testDateGreaterThanOrEqual(){
  Date aday{2012,2,29};
  Date later{2012,3,1};
  ASSERT(later >= aday);
  ASSERT(aday >= aday);
  ASSERT(!(aday >= later));
}


void testDateLessThanOrEqual(){
  Date aday{2012,2,29};
  Date later{2012,3,1};
  ASSERT(aday <= aday);
  ASSERT(aday <= later);
  ASSERT(!(later <= aday));
}
```

Now for the inline implementation of the operators that we put within **class** `Date`'s header file but after the class definition. We mark functions that are implemented within the header file with the keyword **inline** to avoid violating the one-definition rule.

```
inline bool operator>(Date const& lhs, Date const& rhs){
  return rhs < lhs;
}
inline bool operator>=(Date const& lhs, Date const& rhs){
  return !(lhs < rhs);
}
inline bool operator<=(Date const& lhs, Date const& rhs){
  return !(rhs < lhs);
}
```

What is missing until now, is the test for equality **operator**== and inequality **operator**!= or our `Date` values. Even those can be implemented based on **operator**<. If value `a` is neither less than `b` nor greater than `b`, both values must be equal.[11] Implementing **operator**== furthermore allows us to use `ASSERT_EQUAL` from our unit testing framework in case we create more functionality modifying dates. The inequality **operator**!= can then be implemented based on equality likewise.

[11] Alternatively you can say `a == b` when `a <= b && b <= a`.

But first, as you already expect, a test case for our **operator**== . Note that we test against `Date` values, we construct on the fly. However, you might wonder why there are parentheses around the construction of `Date{2012,7,29}`. This is due to the fact, that our testing framework's `ASSERT` is a macro and as such the preprocessor would interpret the commas of the initializer list as separating macro arguments. However, since the `ASSERT` macro only accepts a single argument this would lead to a syntax error. Therefore, we need to place *parenthesis around macro arguments* that contain commas. The

parenthesis around macro arguments

last ASSERT macro call does not suffer, because it already has a pair of parenthesis around the equality expression to be able to apply the **not** operator to it.

```
void testDateEqual(){
  Date aday{2012,7,29};
  ASSERT(aday == aday);
  ASSERT(aday == (Date{2012,7,29}));
  ASSERT(!(aday == Date{2012,7,28}));
}
```

And here goes the implementation of our equality operator, again as an **inline** function in the Date.h header outside the class definition.

```
inline bool operator==(Date const& lhs, Date const& rhs){
  return !(lhs < rhs) && !(rhs < lhs) ;
}
```

And now to complete our set of relational operators for Date the test an implementation of the inequality operator.

```
void testDateInEqual(){
  Date aday{2012,7,29};
  ASSERT(!(aday != aday));
  ASSERT(aday != (Date{2012,7,28}));
}


inline bool operator!=(Date const& lhs, Date const& rhs){
  return !(lhs == rhs);
}
```

This concludes our implementation of a complete set of relational operators for **class** Date. However, the implementation of the equality operator is slightly inefficient, so one option would be to implement it as a member function instead, like **operator**<. Another observation you might make is, that all the additional free function relational operators have a very schematic implementation, but see the next section on how to eliminate such boilerplate code.

Avoiding Boilerplate Code for Operator Functions

You might have already asked yourself, why is code that is so schematic, as the implementation of our relational operators can not be provided through a library. It can, but it is not provided by the C++ standard library (yet). However, the boost operators library[DAFo8] provides a means to implement operators for a type based on some elementary overloaded operator.

inherit

In order to get the operators for greater etc. we need to **#include** <boost/operators.hpp> and let our Date class that implements the less than operator publicly *inherit* from boost::less_than_comparable <Date>. We inherit by adding a single colon after the class' name in

its definition and list all of its base classes after that, just before we open the brace of the class' member definition. Depending on the keyword **class** or **struct** that is used for the class definition the inheritance is either **private** or **public** unless we specify the opposite.

In the given case the inheritance looks strange, because we pass the class we are just defining as a template argument to the library template class. It is also completely different from inheritance that you might have experienced in Java™ or C# . The base class does not provide member functions or member data and there is also no dynamic polymorphism involved but only additional separate globally available operator functions. Because of that, it even doesn't matter which access control tag **private** or **public** we use. The run-time and space cost of this special inheritance is zero. The **inline** definitions of >, >=, <= are removed, because they are now redundant. With that in place, the beginning of the header file `Date.h` looks as follows.

```
#ifndef DATE_H_
#define DATE_H_
#include <iosfwd>
#include <tuple>
struct Date
: private boost::less_than_comparable<Date> {
```

Please note, that I used **private** inheritance to show you the application of the access control tag only. In contrast to the **private**:/**public** :sections within the class definition you do not follow **private**/**public** when specifying inheritanc access control with a colon. The changed code still runs all previous tests successfully.

What remains is the redundant inequality operator and an inefficient implementation of **operator**==. We start out, implementing **operator**== as a member function. Because its logic is simpler we implement it directly within the class.

```
struct Date
: private boost::less_than_comparable<Date> {
//...
  bool operator==(Date const& rhs) const {
    return std::tie(year,month,day) ==
        std::tie(rhs.year,rhs.month,rhs.day);
  }
//...
};
```

Still all tests run, but now, we also should eliminate the boilerplate **operator**!= by inheriting from `boost::equality_comparable<Date>` as follows.

```
struct Date
: private boost::less_than_comparable<Date>
, boost::equality_comparable<Date> {
```

This concludes our excursion on using Boost's `operators.hpp` library. We will come back to it later. While it is wise to provide overloads of relational operators to be able to sort and compare your own types, it hardly ever makes sense to overload arithmetic operators, such as `+, -, *, /`, for your classes unless you are implementing your own numerical types. However, doing so efficiently and correctly requires a lot of skill, especially when such a numerical type needs to interact with the built-in types seamlessly. However, you are encouraged to implement a toy version of a rational numbers type in the exercises.

Output and Input with Shift Operators

After we've got the first taste of operator overloading you might ask why can't we output our `Date` values using the left shift **`operator`** `<<`. Well, you can, by overloading it. Since we already have the formatting function `Date::print` in place, we can do so as an **`inline`** function as follows.

```
inline std::ostream&
operator<<(std::ostream& out,Date const& d){
  d.print(out);
  return out;
}
```

We need to return the stream object, to allow output expressions chaining several values, as you have seen previously, *e.g.*, in the implementation of `Date::print`.

That looked easy, but it was the case, because we already hat the underlying implementation in member function `print`. However, we do not yet have a corresponding input function. But we can look at the necessary function signature to get an impression what we need to implement reading a date from an input stream.

```
std::istream& operator>>(std::istream& in, Date &adate);
```

You can observe that we need to pass the variable to be filled with the date read by reference. We need that side effect, otherwise the function cannot return the value of its purpose. We have to return the passed-in `std::istream&` to ensure that the input operators can be chained.

I always recommend to provide a member function that allows the construction of an object from a stream. Then the implementation of **`operator`**`>>` can rely on that member function without needing to care further about the internals of the class. How do we design such a reading/constructing member function. There are several options:

- Provide a regular member function `read(std::istream&)` that modifies the **`this`** object if the input is valid.
- Provide a constructor `Date(std::istream&)` that initializes an object from the input.
- Provide a *factory function* `Date make_date(std::istream&)` creat-

factory function

ing a date object from parsing input.

Each of these functions provide different challenges and options on what to do, when the input format is wrong. A `read` member function can set the stream to its failing state when the input is wrong, the other options either have to throw an exception or provide a default value, when no valid date is input. Otherwise they would create a `Date` value with the invariants violated. Because the typical behavior of an input operator on failed input is to set the stream to an invalid state and not throw an exception I recommend to you to emulate the first option for your future classes requiring input, because it allows a straight-forward delegation from **`operator`**`>>` symmetric to our output operator.

```
inline std::istream&
operator>>(std::istream& in,Date& d){
  d.read(in);
  return in;
}
```

This approach only fails, if there is not reasonable or fast means to default construct an instance of your class, which is required for initializing the variable before you call the `read()` member function. In that case, a factory function throwing an exception if there is an input error might be better, but you loose the ability to combine reading your type together with others in a single statement.

Dealing with Input Errors Numeric input on a `std::istream` will set the variable to zero, if the input can not be converted to a number or to the variable's numeric type maximum or minimum value if the read-in value is out of the type's bounds. For example, if you input `"70000"` when an **`unsigned short`** `x;` variable is to be read like `std::cin >> x;` then x will get the value `0xFFFF (65535)` the maximum number representable in the type **`unsigned short`**.[12]

We need to consider if we want to consider a similar behavior for reading date values as well. This means to decide what values to take for date "zero". We already have defined a minimum and maximum representable date, because of or invariants. Either of those might make a good candidate as a default value for failed input. However, the 31.12.9999 is much easier to remember than New Year of 1813 and also well out of the range of dates we will encounter.

Implementing `read()` allows us to skip that decision and just leave the date variable it is applied on unchanged in case of an input error. We can employ our `isValidDate()` static member function to figure out, if the input is correct. However, a good input routine should be flexible, so that different date formats can be accepted. While not changing the sequence of day-month-year, we can accept different punctuation characters and optional spaces in between. Also the possibility to enter the year in two digit form if in the 21st century can be handy.[13]

[12] Assuming `unsigned short` is represented in 16 bit.

[13] This will give us a Y2.1K problem, but I do not want to introduce obtaining the current year yet, so that we can have a floating window of dates.

Testing Input This allows us to create a number of positive test cases for testing `Date::read()`. First with 4 digit years:

```
void testDateReadValidDates(){
  std::istringstream
       is{"17.08.2012 18/8/2012 19-08-2012"};
  Date aday{2000,1,1};
  Date expect{2012,8,17};
  aday.read(is);
  ASSERT(!is.fail());
  ASSERT_EQUAL(expect,aday);
  expect.nextDay();
  aday.read(is);
  ASSERT(!is.fail());
  ASSERT_EQUAL(expect,aday);
  expect.nextDay();
  aday.read(is);
  ASSERT(is.eof());
  ASSERT_EQUAL(expect,aday);
}
```

After each read, we test the state of the stream to ensure that input succeeded by calling `std::istream::fail()` which would return true, if the input format could not be converted. At the end of the input we check that the end was actually reached. [14]

[14] Please note that such large tests with multiple assertions are considered bad practice. It would be better to split the test into three functions.

If we do consider 4 digit years, we can also think of interpreting input in the YYYY-MM-DD form. We do not intend to represent dates around the years Jesus Christ lived, so it is unambiguous if the first number read is actually a valid year to class `Date`'s definition.

```
void testDateReadYYYYMMDD(){
  std::istringstream is{"2012-08-20"};
  Date aday{2000,1,1};
  aday.read(is);
  ASSERT(!is.fail());
  ASSERT_EQUAL((Date{2012,8,20}),aday);
}
```

On the other hand it is very convenient to have only to input 2 digits for the year. But we only allow that in the format of DD.MM.YY, because otherwise we are not be able to distinguish it from YY-MM-DD up to year 2032. Therefore, we test for two digit years representing years in the 3rd millenium.

```
void testDateReadShortYear(){
  std::istringstream is{"17.08.12 1-9-13"};
  Date aday{2000,1,1};
  aday.read(is);
  ASSERT(!is.fail());
  ASSERT_EQUAL((Date{2012,8,17}),aday);
  aday.read(is);
  ASSERT(! is.fail());
  ASSERT_EQUAL((Date{2013,9,1}),aday);
}
```

Now we should test for actually failing conversions such as an input of "1-1-100" (wrong year), "a.b.2000" (not numbers), "1:1:11" (wrong separator). However, there are so many things that can go wrong, that all useful tests for those cases would easily get boring. If reading fails, the variable should keep its original value. Please note, that for testing the case of reading after a failed read,*i.e.*, trying to read the day of month in the second case, the stream will already be in a fail state and the invalid character(s) will stick in the input stream. To compensate for that our test case reads characters up to the next white space character to be able to continue with the next read.

```
void testDateReadOutOfRangeDateKeepsOriginalValue(){
  std::istringstream is{"1-1-100"};
  Date aday{2000,1,1};
  Date const expect{aday};
  aday.read(is);
  ASSERT(is.fail());
  ASSERT_EQUAL(expect,aday);
}


void testDateReadInvalidNumbersFailAndContinue(){
  std::istringstream is{"a.b.2000 1:1:11"};
  Date aday{2000,1,1};
  aday.read(is);
  ASSERT(is.fail());
  is.clear();
  std::string dummy;
  is>>dummy; // skip up to ws
  aday.read(is);
  ASSERT(is.fail());
  ASSERT(is.eof());
}
```

As you see parsing input robustly and checking for that robustness can be a messy business, but it is necessary. In production quality systems one often separates the two problems of input syntax checking and value conversion. The former can be done on a character and string basis, may be using regular expressions (section **??**) or even a grammar-based parser. Only when the syntax is OK, value

conversion happens, that now no longer has to deal with potential errors.

Implementing Input We implement `Date::read()` less than perfect but just attempting to read three **`int`** values separated by an arbitrary non-whitespace character. Only after that, we check if everything is OK.

```
void Date::read(std::istream& in) {
  int in_day{}, in_month{}, in_year{ };
  char sep1{}, sep2{};
  in >> in_day >> sep1 >> in_month >> sep2 >> in_year;
```

To allow a user to either enter the date YYYY-MM-DD or DD.MM.YYYY we check if the first number input is actually a valid year, because that is beyond the range of possible day values, we swap the values of `in_day` and `in_year` for further processing. While this may be accompanied by a variety of means, using the library function `std::swap` is not only efficient, but also tells exactly what will be happening.

```
  if (isValidYear(in_day)) std::swap(in_day,in_year);
```

To be able to employ our `isValidDate()` check, we first adjust variable `in_year` if it is representing a shorthand year for the 21st century.

```
  if (in_year >= 0 && in_year < 100)
    in_year += 2000;
```

Now we are ready to check if everything is OK, we start out to check if some reading of the numbers went wrong and if the two separators are the same character. We also make sure that the separating character is one of those we defined. And last we also ensure that the read values actually conform to a valid date.

```
  if (!in.fail() && sep1==sep2
    && (sep1=='.' || sep1=='/' || sep1=='-')
    && isValidDate(in_year, in_month, in_day) ) {
    year=in_year; month=in_month; day=in_day;
```

If something went wrong, we need to signal it through the streams `iostate` flags, by setting the bit `std::ios::failbit`.

```
  } else {
    in.setstate(std::ios::failbit | in.rdstate());
  }
}
```

A Parsing Constructor For completeness and also to show some other aspects as well, let us implement a further constructor constructing a date from reading a stream. Because we've already implemented `read()` we can rely on it in the constructor's function body. However,

there is the need to initialize its members first. Here we provide default initialization for the member variables, even though that the invariant of our class would be violated (0.0.0 is not a valid date). However, because we are only returning from the constructor with a successfully established invariant, everything is fine up to now. Note, only because our member function `read` does not rely on `Date` 's class invariant, we can call `read` from the constructor. In general such practice can be dangerous, so try to avoid it. An option would be to implement the reading and parsing in the constructor and having `read` delegate its work to such a constructor. But that would have meant to create a superfluous object when reading from the stream and `std::swap`ing it with the **`this`** object. This can also be a problem when such an object would be very big and expensive to construct.

```
Date::Date(std::istream& in)
:year{},month{},day{} {
  read(in);
  if(in.fail())
    throw std::out_of_range{"invalid date"};
}
```

After extending class definition in the header file accordingly, I went on to write a test case for the stream operator, but I made a typo as follows when trying to input `adate` from the stream:

```
is > adate;
```

Note that I used *greater than* but not *right shift*. The code compiles. Why? We have met one of C++'s features that is both heaven and hell. We have overloaded `>` to take two `Date` objects as arguments. And we have a constructor that "converts" a `std::istream` object into a `Date` object. By asking the compiler to "compare" a stream object with a date, the compiler applies this *conversion constructor* automatically without us asking and actually compared the read in date and the variable instead of putting the value into the variable. With numerical types, this automatic conversion can be convenient, such as adding `5 + 3.14` (**`int`** plus **`double`**) but for most of our own types such automatic conversion constructors are not a good idea. For example, when you have `std::cin > adate;` in your program, it will actually wait for you to input a valid date, but that will never end up in the variable adate. I've been debugging and tracing my code for a long time before I've spotted such an error in the past. Fortunately, C++'s automatic type conversion only applies to situations where a value can be converted to a required type in one step. There is never a longer sequence of conversions applied to a value, even though it might be possible without doing the type conversions explicitly, *e.g.*, through **`static_cast`**`<atype>(avalue)`.

conversion constructor

The language provides a means to prohibit such automatic conversions: the keyword **`explicit`**. If we put that in front of our constructor declaration, it will never take part in such automatic conversions, and must be used "explicitely" to create a `Date` from a `std::istream` object. You do not repeat **`explicit`** before an out-of-class definition

explicit constructor

of the constructor declared as **explicit**.

So our constructor's declaration in Date is as follows.

```
explicit Date(std::istream& in);
```

Caution 4.1: Define Constructors explicit

Avoid automatic type conversion with explicit

Automatic type conversion can be both a curse and a blessing. However, in most cases you want to avoid it, so define your single parameter constructors and type conversion operators as explicit, unless you intentionally want to combine different types in an expression and conversion can occur without side effects on the origin or without hidden cost such as a large allocation.

Factory Function To complete the range of options we implement a factory function creating a Date value by reading from a stream. This factory function returns a default value, when the input fails instead of throwing an exception. I have chosen the largest representable date to be such a default value in case of the error. It is improbable that our software will need that date as an operational value. Boost's class boost::date [Gar05] provides special exceptional values like not_a_date_time for such purposes, but we do not, to keep things simpler.

```
Date make_date(std::istream& in)
try {
  return Date { in };
} catch (std::out_of_range const &) {
  return Date { 9999, 12, 31 }; // default value
}
```

Factory functions are often named with the prefixes make_ or create_ . If they require detailed access to a class' internals you can define them as static member functions ("class methods"), but in our case it is not necessary, so we declare it as a separate function side-by-side with our class.

Please note that in the success case we construct a Date object from the stream. Because our constructor is explicit, we can not use implicit conversion here and have to name the type we construct.

Placing of Functions associated with a Class

With most overloaded operators that you might want to define for your types, you have the choice to either implement them as member functions or outside of your class as regular functions.[15] In addition you need to decide if you put a function's definition in the

[15] See Table 1.3 and Table 1.5 for exceptions, such as assignment, function call and index access operators.

header as an **inline** function or declare it only there and keep the implementation separately in a `.cpp` file.

inline function

Defining a function completely in the header as an *inline function* should be done if it is very short. This can improve performance, because it allows the compiler to optimize away the function call overhead and through inlining the necessary statements can open up the ability to further optimizations. Some such functions might even be completely eliminated at runtime. As a rule of thumb you should define operator functions in their header that delegate their work to other (operator) functions only, such as the relational operators and the I/O operators of our class `Date`. If such a function definition is in the header it must be marked as an **inline** function, if it is not defined within a class' body.

When designing a class you define the interface of its instances to the outside world through its public non-static member functions. However, sometimes a function taking a class instance as its first parameter doesn't need to access its private members. Then the question is open, if you should implement such a function still as a member function, or as a separate function outside of the class. A function that is not participating in dynamic polymorphism and doesn't need access to private members can be defined as a non-member function outside of the class of its first parameter (which would be implicit if defined as a member function). Such non-member functions often reduce coupling, because they only rely on a class' public interface. However, if your later change your mind and your function is not an operator function you have to change all calling sites which can be a large editing effort.[16]

[16] There are thoughts about relieving the need to call member functions with the dot notation `a.foo()` as `foo(a)` and vice versa in a future C++ standard, but nothing has been decided or even specified at the time of this writing.

A further recommendation, we haven't followed yet with class `Date`, its separate operator functions, and functions using it as parameter type, is to put the class and all functions using the class type as parameter type in one namespace. This actually allows the compiler to determine which function to call. An operator function as well as a regular function is searched in all namespaces that its arguments belong to while reducing the chance for ambiguities. For overloaded operator functions this is especially important, because you want to write `a > b` and have the compiler figure out it should call **operator**`>(a,b)` that suits the types of `a` and `b`. This kind of function determination based on the namespaces of the arguments at a call site is called *argument dependent lookup* or *ADL*. Our header file `Date.h` is adjusted accordingly as follows. You need also adjust the implementation file `Date.cpp` accordingly, by either prefixing the function names with `date::` each, or by putting them in the namespace there as well (see section 3.2). Also all non-member factory functions returning our type belong in its surrounding namespace to avoid naming conflicts.

argument dependent lookup

```
#ifndef DATE_H_
#define DATE_H_
#include <iosfwd>
#include <tuple>
#include <boost/operators.hpp>
struct Date
: private boost::less_than_comparable<Date>
, boost::equality_comparable<Date> {
//...
};
inline std::ostream&
operator<<(std::ostream& out,Date const& d){
  d.print(out);
  return out;
}
Date make_date(std::istream& in);
//...
} // namespace date
#endif /* DATE_H_ */
```

4.4 More on Class Initialization, Internals and Destruction

On a first read, you might skip this section, because it delves into details that usually do not have to bother you. I give a brief summary of what you can expect.

We haven't seen all potential means of initializing class instances yet. There are compiler automatisms you should rely on, suppressing them should be left to library classes written by experts. We haven't heard about the default constructor. You can provide data member initializers new to C++11, instead of repeating stuff at constructor's initializer lists. And, we haven't really met **this** yet, even though that actually doesn't belong under this topic.

Rely on Automatisms when defining Classes

copy constructor

When looking closely at above usages of our `Date` data type, you might have seen, we have used it in operations, we actually haven't defined ourselves. For example, we could return `Date` values which actually involves copying its objects, or we can initialize a `Date` variable from another `Date` value. Also assignment would have been possible. This means that without any specific instructions the compiler provides our class with a so-called *copy constructor* and an assignment **operator=**. Their signatures are

```
Date(Date const& other);
Date& operator=(Date const& other);
```

The compiler generates them with an implementation of memberwise copy. Older C++ books often encourage you to implement these member functions and additionally also a destructor, which would be called whenever a `Date` value's lifetime ends.

```
~Date();
```

I strongly recommend that you design your classes in a way that the compiler-provided defaults for these functions are perfectly suitable. You will learn throughout this work what tools the language and its library provide to actually make your classes that way. User-defined versions of copy- or move- constructors and assignment operators as well as destructors are usually only needed for library classes that require their own resource management.

For documentation reasons and also in very special expert-level situations you can declare these "special" member functions in your class and say that you want the compiler-provided default implementation by putting =**default** after their signature.

```
Date(Date const&)= default;//do not do this
Date& operator=(Date const&)= default;//do not do this
~Date()= default;       //do not do this
```

I recommend to refrain from such practice as well, because it not only adds clutter to your code, if you provide it for documentation reasons only, but it also hinders the compiler to automatically move-enable your class which it will only do if you do not declare any of the above class members. The implicit move constructor can have a huge influence on performance of your system when your class includes move-enabled data members, such as `std::vector`, and it's objects are copied/returned often.[17]

If you insist on documenting the default versions of your copy constructor and assignment operator, you should provide declarations for move constructor and move-assignment operator as well as defaulted.

```
Date(Date &&)= default;   //do not do this
Date& operator=(Date &&)= default;//do not do this
```

[17] There are special cases, where automatic declaration or definition of the special member function is suppressed. It requires language experts to actually memorize and apply those rules correctly without deep thinking. You are better off, if you write your code in a way that you do not need that knowledge.

What is a Default Constructor?

A default constructor is used when you define a variable and do not provide arguments to its initializer, or even do not put any initializer at all.[18] For example, as in the following statements.

```
Date adate{};
Date bdate;//not recommended!
```

If you try that with the current version of our Date class, you get a compile error. You would have been able to do so with the initial trivial version of **class** `Date` on page 100. The compiler did automatically provide a constructor with the following signature, because no other was defined.

```
Date();//declaration made in class Date by the compiler
```

[18] This is dangerous, because for trivial types this means it can remain uninitialized, but that won't be the case for class `Date`.

However, once we define our first own constructor to ensure the class invariant, we lose the compiler-provided default constructor. The reasoning behind that, is that any user-declared constructor might impose an invariant that a compiler-defined default constructor might

not guarantee.

We can resurrect a default constructor by defining one that doesn't require any arguments. Either by declaring and defining one as above, like

```
Date(): year{},month{},day{}{}
```

which would not guarantee our invariant of a valid date, because it initialized the member variables with zero as would be the resurrection by asking the compiler for its default implementation.

```
Date()= default;
```

To establish our invariant we need to consider what a good default value would be. We already defined that when providing the last day of year `9999` as a special value denoting an input error in our factory function `make_date`. With that we can define a default constructor as follows.

```
Date::Date()
:year{9999},month{12},day{31} {
}
```

This allows us to adjust our factory function to avoid duplication as follows.

```
Date make_date(std::istream& in)
try {
  return Date { in };
} catch (std::out_of_range const &) {
  return Date{};
}
```

Old code uses parenthesis for initializers

In pre C++11 code you will find round parentheses around the member initializer values in a constructor definition as well as in calls to the constructor when providing the initial values. For example,

```
Date(): year(),month(),day(){}
Date adate(2012,8,23);
```

However, when you try to define a variable with by using its default constructor, you need to omit parenthesis, otherwise it is parsed as a function declaration, which can lead to interesting compiler error messages.

```
Date defaultDate;
Date noVariableButAFunction();
```

Whenever you can use them in C++ stick to curly braces for initializing a value.

Member Initializers and Delegating Constructors

When you have a class with many constructors, you might end up with repeating the initializers for members a lot. This can be tedious. There are two features of C++11 that can help to alleviate that situation and reduce such initializer duplication. A third option available already with earlier C++ standards is to rely on the default initialization of those member variables that you get for free when the class is not a *trivial type* and if the default value, such as zero for numbers or an empty container or `std::string` for those types is what guarantees your class invariant. [19] Unfortunately that option doesn't suite our class `Date`, because of its stricter invariant.

trivial type

[19] A trivial class compares to a C `struct` and only has members of fundamental or trivial type and no user-provided constructors amid other rules.

The first C++11 feature to reduce duplicated member initializers at constructor definitions, is called *non-static data member initializer* (aka *NSDMI*). Each such member variable definition can include an initializer as a regular variable. The value in that initializer can not depend on any constructor parameter, that means you can usually only specify a constant value as the initial value. Whenever a constructor omits such an initialized member variable from its own initializer list the member initializer provides its initial value. However, even if you specify an initializer for all member variables, you need to define a default constructor, when you also provide other constructors. Our Date class can contain the following initializers for the default date.

non-static data member initializer

```
struct Date
: private boost::less_than_comparable<Date>
, boost::equality_comparable<Date> {
  Date()=default;
  //...
private:
  int year{9999};
  int month{12};
  int day{31};
//...
};
```

The alternative C++11 option for saving duplication in initializer lists are *delegating constructors*. The also solve the problem of duplicate base-class initializers that are preceding the member initializers in a constructor's initializer list. Our class `Date` doesn't have this problem, but still I show you the syntax. Instead of a base class or a member initializer, you just provide a call to the other constructor you rely on. In the following example this is `Date`'s default constructor.

delegating constructors

```
Date::Date(std::istream& in)
:Date{} {
  read(in);
  if(in.fail())
    throw std::out_of_range{"invalid date"};
}
```

Looking at `this`

I have mentioned the **this** object several times already. If you have programmed in any object-oriented language you are aware of that implicit parameter that a method/member function gets. Some languages use the name `self` as a name to refer to that implicit parameter of object methods. In most other object-oriented languages you can use **this** like a regular variable that you can call methods on by using the dot notation, *e.g.*, in Java™ you can write **this**`.toString()`.

For historical reasons, C++'s **this** is a bit different. Instead of being a reference to the object given as a member-functions implicit parameter it is a *pointer*. I won't explain you the details of pointers right now, because you don't need other pointers than **this**. But I show you the syntax to access a member using the *this pointer*.

this pointer

To access a member of your class through **this** you write

```
this->member
this->memberfunction()
```

That syntax is only required in cases where a local variable has the same name as one of the class' members, for example in a member function body, where you named a parameter like a member variable you need to access. There are special cases in **template** classes or member functions where using **this**`->` can also be relevant.

Another more common situation where you need to de-reference the **this**-pointer is when you want to pass the current object as an argument to a function you call or in an expression. Then you use the prefix dereference **operator**$*$.

```
foo(*this);
```

An application in our class would be to implement (against my suggestion) the inequality operator ourselves as a member function. This implementation looks as follows.

```
bool Date::operator!=(Date const& rhs) const {
  return !(*this == rhs);
}
```

You can even access a member by using $*$**this** but before you can apply the dot operator you have to put parentheses around `(*this)`, because according to Table 1.3 dot has a higher precedence than prefix $*$: `(*this).memberfunction()`

Clean up and Lifetime Tracing

This section shows you how to define a destructor for your class. Normally there should never the need to define your own destructor like with the copy/move constructors and assignments. However, for learning the scoping rules, it can be a handy tool to define a class that actually outputs something from its constructors and destructors, so that you see, when an object is actually created, copied, and destroyed. Try to figure the output of program tracer.cpp (Listing 4.1) first and check your understanding with running it. Having

```
#include <iostream>
#include <string>
struct Tracer{
  explicit Tracer(std::string name="")
  :name{name}{
    std::cout << "Tracer created: " << name << std::endl;
  }
  ~Tracer(){
    std::cout << "Tracer destroyed: " << name << std::endl;
  }
  Tracer(Tracer const& other)
  :name{other.name+" copy"}{
    std::cout << "Tracer copied: " << name << std::endl;
  }
  void show() const {
    std::cout << "Tracer: " << name << std::endl;
  }
  std::string name;
};
void foo(Tracer t){
  Tracer trace("foo");
  t.show();
}
Tracer bar(Tracer const &t){
  Tracer trace("bar");
  t.show();
  return trace;
}
int main(){
  Tracer m("main");
  {
    Tracer inner("inner");
    foo(inner);
    auto trace=bar(inner);
    trace.show();
    inner.show();
  }
  foo(Tracer("temp"));
  m.show();
}
```

Listing 4.1: tracer.cpp

a well-defined moment when objects are destroyed is a character-

izing feature of C++ that distinguishes it from inherently garbage-collected languages like Java™ and C# . This allows classes to deliberately provide resource management of external resources. By acquiring a resource in constructors and releasing it in the destructor of a class, one can guarantee that clean up is already done correctly, even in the case of exceptions being thrown. The underlying pattern is called *RAII*: *resource acquisition is initialization*. Applying RAII correctly also means that one needs to take care what happens if your class is copied or moved. You can either prohibit both, or enable move only, to make sure that only one object actually is in charge of your resource. If you copy such a resource management class then you need to take care that the last copy holding onto your resource is actually cleaning it.

RAII

In the past, C++ tutorials spent pages and drill to teach you how to do this right. While you still need to grasp the basics of object lifetime, with C++11 it is no longer necessary that you need to write such complex and error prone boilerplate code with copy constructors, assignment operators and destructors. Follow the rules that I am giving you and let the compiler do all necessary steps for you. In the case you want your class to actually apply RAII to a special resource you use the library classes `std::shared_ptr` or `std::unique_ptr` to wrap such resources. While the names imply thinking of "pointer", the resource these classes can manage doesn't need to be a classic pointer type allocated from free store (see section **??**). There are even proposals to add further resource management RAII-wrapper types to a future version of the C++ standard library.

To sum up, whenever you are tempted to write your own version of a copy/move constructor, a copy/move assignment operator, or a destructor for your classes, hold on, and think if and how the compiler provided defaults can be employed.[20]

[20] There are a few exceptions, where you have to define a destructor when using a `std::unique_ptr` but I will show you later.

Value Classes versus Object Classes with Inheritance

If you are experienced with object-oriented languages like Java™ , you might be asking yourself how to build your own class hierarchies. We will come to that, but using inheritance and polymorphism is not the only purpose for defining classes. In contrast to Java™ , C++'s classes can be used to construct types representing values, where Java™ 's objects are always represented by references (excluding some basic types as int and char). The compiler provides automatisms of copying and assigning instances of classes in C++ and thus treats these instances as values like the built-in types, **`int`**, for example. If nothing special is done, such class instances can be created as temporaries and local variables and are expunged automatically when their clearly defined lifetime ends. With the help of operator overloading, you can make your type even behave like a built-in numerical type and through inlining of these functions the resulting code becomes as efficient as using built-in types directly. But in contrast to those, you will get better type checking and can

prohibit unwanted automatic conversions, which might happen, for example, between **double** values and **int** variables.

On the one hand, this is a great benefit of C++: You can create class types that work as efficient and syntactically identical to its built-in types. There is no need to create instances of these types on the heap, as you would have to do with all class types in Java™ . And the deterministic model of cleaning up after a value's life-time expired also allows to work with such value types easily.

On the other hand, may habits that you might have already acquired in Java™ , C# , or from "old-school" or bad practice C++ programs are no longer beneficial for a modern lucid style of C++.

The concept of values goes even further than simple classes as our class `Date`. Also `std::string` and the containers of the standard library work like values, if their elements act like values. Even though, in the case of large data structures heavy copying can be negative on the performance, but then you can pass those elements down the call chain by reference. Returning them by value is fine, since the compiler will automatically elide copying or use their move-constructors to establish efficiency. There are old books or articles explaining how to avoid expensive copies, but their advice is no longer true.

With C++'s feature of move-constructors and move-assignment operators a new dimension is added. With those you can even build classes that almost behave like values, even though they need to have a uniqueness about them. For example, we learned, that we can not copy stream objects. However, we have seen that we can pass a file stream object out of a factory function, because it is a move-only type (see page 81). Such a file stream needs to be unique, so that when its life-time ends, the external resource it represents—the open file—gets cleaned up and closed, otherwise your program might run out of available operating system resources for open files when running long enough.

4.5 *Enumeration Types Representing the World*

You might have observed that in our date examples so far, we encoded months as numbers. We also observed, that it might be hard to distinguish a month from the day in the months, especially when you are used to another sequence of day, month and year than those provided by our constructors. For data types that have such a few values, like the months of a year, the days of the week, the colors of a traffic light, or the states in a (small) state machine model C++ provides the type construction possibility with **enum**. For example, a data type for the days of the week can be defined as follows.

```
enum day_of_week { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
```

This introduces the new type `day_of_week` as well as the abbreviations of the weekdays as identifiers into the current scope. The individual enumeration values are represented as integer constants and their numbering start with zero. They are compatible with the

type **int** and can be assigned to integral variables.

```
int day=Fri; //initializes with 4
```

On the other hand, if you define a variable of type `day_of_week` you can not assign to it an integral value, only the pre-defined constants.

```
day_of_week aday{Tue};
aday = 5; //doesn't compile
```

While this safeguard is good, it is not fool proof. For example, when you want to compute the next day, you can only to that by integer arithmetic. The resulting value has to be **static_cast**<day_of_week> to allow the compiler to assign the value again to our variable `aday`. But for performance reason the compiler will not ensure that the integer value actually corresponds to one of the encoded values of the enumeration. For example, the following code just compiles fine.

```
aday = static_cast<day_of_week>(42);
```

Without further specification, the individual enumeration identifiers get assigned subsequent positive integer values beginning with zero. However, using an assignment operator and a constant, we can define either a starting position or each individual value. This allows us to define an enum-type for our twelve months that starts with 1, like the calendar will number our months.

```
enum Months {
  jan=1, feb, mar, apr, may, jun,
  jul, aug, sep, oct, nov, dec
};
```

Scoping of Enumeration Identifiers

One problem with the "classic" style of enumerations is, that the identifiers introduced as enum value identifiers are scoped in the surrounding scope. For example, if the **enum** `Month` of above would be defined on namespace-level scope, the identifier `dec` for December would conflict with the use of that name for anything else, e.g., a function `dec()`.

One means, also working older C++ would be to put the enum definition within a class scope. For example, we could define the **enum** `Months` within our **class** `Date` and then, outside of that class one has to qualify the month names by that class, *e.g.*, as `Date::jul`. Current C++ provides so-called scoped enumerations, that actually nest the enumeration constants within a scope denoted by the enumeration type, *e.g.*, `month::aug` when defined as below.

```
enum class month {
  jan=1, feb, mar, apr, may, jun,
  jul, aug, sep, oct, nov, dec
};
```

Therefore, always define enumeration types either in class scope, when they are related to a specific class or by using a scoped enu-

meration, or both so that the identifiers of the enumeration constants aren't "polluting" the surrounding scope.

It is not required that the enumeration constants refer to unique values. For example, we can extend our enum type to spell out the month names in addition to the three letter abbreviations. We even do not need to explicitly assign each value, but can use the fact that subsequently defined enumeration constants get the value of the previously defined constant incremented by one.

```
enum class month {
  jan=1, feb, mar, apr, may, jun,
  jul, aug, sep, oct, nov, dec
  , january=jan, february, march, april,
  june=jun, july, august, september, october,
  november, december
};
```

Please note, that since May already is the same as its abbreviation we have to omit it from the second list of fully spelled months and reset the compiler internal enumeration counter by assigning the value of `jun` to June.

Defining the Space used by Enumeration Variables

The problematic issue of potentially invalid enumeration values and their use in integer expressions can, in some cases, also be an advantage. For example, there are uses where enumeration values represent binary flags that can be usefully combined by bitwise operators. This is especially true in programs for embedded systems, where enumerations can be used to represent bits in a register. Or, when several options are available for a "bitmask" type, that can be combined.

However, it used to be bad practice to use enumeration types as members in aggregate types such as structs or arrays for embedded systems, because the actual representation, *i.e.*, the amount of space allocated to a variable of the enumeration type depended on the compiler and the number and range of actual enumeration values defined. For "classic" enum types the compiler is free to chose any appropriate integer type that can fit all the values, but you might not know which is chosen when you have to port your code or binary data. Therefore, C++11 introduced the ability to deliberately specify the type that should be used for the internal representation of the enumeration type. Like a super class for class types, you can use one of the integral types as the base type of your enumeration.

```
enum class launch_policy : unsigned char {
  sync=1, async=2, gpu=4, process=8, none=0
};
```

This enumeration base type variant is mostly relevant for embedded systems where space is tight and one needs tightly packed data

structures with defined and well-known sizes. A second use for these types is when the individual values are to be internal only, but the type might be used, *e.g.*, for defining a member variable of a class. Then the base type allows to declare an enum type for such a variable as an *opaque enum type* and hide the individual values used if they are not required outside of the class' implementation. For example, a class representing a state machine can use an enumeration type for the individual states without exposing the actual states used. This can be very handy, if you want to use an enumeration and want to shield the users of your class from its further internals. It also allows you to extend and modify the states of the state machine without affecting the users of your code. Please see Statemachine.h (Listing 4.2) for the forward declaration of the opaque enum type `Statemachine ::State` and then in Statemachine.cpp (Listing 4.3) for its definition and usage. The code uses a **`switch`** statement. This is one of the rare cases where **`switch`** statements are useful, but there are other options, such as a lookup table with lambdas or function objects that can be used, once the state machines get more complicated.

opaque enum type

Listing 4.2: Statemachine.h

```
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

struct Statemachine {
  Statemachine();
  void processInput(char c);
  bool isDone() const;
private:
  enum class State : unsigned short;
  State theState;
};

#endif /* STATEMACHINE_H_ */
```

Overloading Operators for enum Types

Because enumerations are distinct types, we can overload operators for enumeration types as well. The only difference is, that in contrast to class types, all overloaded operators must be implemented as regular functions. We can not define member functions for enumeration types.

So if we change our class representing dates to actually use an enumeration for months, we can overload the output operator for months to get a nicer representation instead. For convenience, we put our **`enum`** `Months` in the namespace of our class, but not within the class. We also refrain from defining it as a scoped enumeration. This avoids too many levels of nesting, when one needs to qualify a month in code using our date library, *e.g.*, `date::aug` instead of `date::Date::Month::aug` in the deepest possible scope nesting.

Listing 4.3: Statemachine.cpp

```
#include "Statemachine.h"
#include <cctype>
enum class Statemachine::State: unsigned short {
  begin, middle, end
};

Statemachine::Statemachine()
:theState{State::begin} {}

void Statemachine::processInput(char c){
  switch(theState){
  case State::begin :
    if (! isspace(c))
      theState=State::middle;
    break;
  case State::middle :
    if (isspace(c))
      theState=State::end;
    break;
  case State::end : break;// ignore input
  }
}
bool Statemachine::isDone()const{
  return theState==State::end;
}
```

```
namespace date {
enum Month{
    jan = 1, feb, mar, apr, may, jun,
    jul, aug, sep, oct, nov, dec,
    january = jan, feburary, march, april,
    june = jun, july, august, september, october,
    november, december
};
std::ostream& operator<<(std::ostream& out, Month m);
```

To implement the output, we define a constant sequence of constants to be used as a local variable of our shift-operator implementation. In addition we declare it as **static** to avoid initializing the constant every time the function is executed. Such a look-up map is often more elegant than a corresponding laborious **switch** statement.

```
std::ostream& operator <<(std::ostream& out, Month m) {
    static std::vector<std::string> const month_map { "Jan",
        "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
        "Sep", "Oct", "Nov", "Dec" };
    out << month_map[m-1];
    return out;
}
```

It would also be convenient, if we should advance the month in

a way that after December the succeeding month is January and not some value not defined in our enumeration. This can be achieved by overloading the increment **operator**++. This operator can usually only be applied to a variable, because its main task is a side effect, actually changing a value, while secondary it also returns a value, in general. Following the rule *"when in doubt, do as the ints do"*[Mey05], we return the new value, when ++ is used in prefix mode and the old value when ++ is used as a suffix to a variable of type `date::Month`. To achieve the side effect, we have to define the parameter as a non-const reference parameter.

You might already wonder, how do we distinguish that difference in functionality. Well the C++ inventor of operator overloading thought of that and defined a special function signature for the postfix version of the increment and decrement operator functions: it has an additional parameter of type **int** that get's implicitly called when we use the operator in a postfix notation. Without that additional dummy argument, the prefix operator is overloaded.

We implement the prefix-version of the increment operator for our type `Month` as follows. When we reach December, we just skip back to January by using a modulo (%) operation. Because our months enumeration starts with 1, we later have to add that one again. Because such integer arithmetic results in an **int** value, a conversion back to `Month` is required.[21]

[21] The conversion must use parentheses instead of braces.

```
Month operator++(Month& m) {
    m = Month((m % 12) + 1);
    return m;
}
```

And to show you the special syntax of the postfix version of **operator** ++ have a look at the boilerplate version that relies on the definition of the prefix version.

```
Month operator++(Month& m, int) {
    Month old = m;
    ++m;
    return old;
}
```

For class types, that you intend to provide with both versions of increment or decrement operators there exists the base classes `boost::incrementable<T>` and `boost::decrementable<T>` in the Boost.Operators library[DAF08] that automatically provide the suffix version, if the class defines the prefix operator overload.

4.6 Defining an Arithmetic Type

First of all, you should only define your own arithmetic types, if you really know what you want to do with it. While it sometimes seems convenient to allow adding an integer to some class type you invented, this might confuse later programmers seeing your code, since

your own version of '+' for your type could do something obscure instead of plain addition. As an example, we will later relate to as well, I will show you how to implement a simple arithmetic type that computes addition and multiplication within a limited range of values defined by the remainder of integral division. This style of modular arithmetic has very interesting mathematical properties that I won't discuss here[SR14]. We implement a class `Ring5` that implements addition and multiplication modulo five. Such a structure is called a ring in mathematics. In that ring, for example, $3 + 4 = 2(7\%5 = 2)$ and $3 \times 4 = 2$.

A first test case allows us to establish the basic infrastructure for that class, a constructor taking an **`unsigned`** value and a `value()` member function for accessing the value. Following our guideline, we start out and declare the constructor explicit.

```
void testDefaultCtor() {
    Ring5 v{};
    ASSERT_EQUAL(0,v.value());
}
void testValueCtor(){
    Ring5 four{4};
    ASSERT_EQUAL(4,four.value());
}


struct Ring5
{
    explicit
    Ring5(unsigned x=0u) : val{ x % 5 } {}
    unsigned value() const { return val; }
private:
    unsigned val;
};
```

Because we already know how most of the operations should be implemented, I will only show a subset of the necessary test cases here and leave the rest for you to implement as an exercise. It is even possible to provide test cases for all possible value combinations. To enable use of `ASSERT_EQUAL` we have to define **`operator==`** and the output operator for our type to be able to interpret potential error messages if that fails.

```
// operator==, operator<< for failures
void testValueCtorWithLargeInput(){
    Ring5 four{19};
    ASSERT_EQUAL(Ring5{4},four);
}
// define operator<< format
void testOutputOperator(){
    std::ostringstream out;
    out << Ring5{4};
    ASSERT_EQUAL("Ring5{4}",out.str());
}
```

The additions to our `Ring5.h` header file.

```
#include <iosfwd>
struct Ring5
: boost::equality_comparable<Ring5>
{
//...
    bool operator==(Ring5 const &r) const {
        return val == r.val;
    }
};
std::ostream& operator <<(std::ostream& out, Ring5 const& r);
```

Here is the implementation file for providing the output operator's definition.

```
#include "Ring5.h"
#include <ostream>
std::ostream& operator <<(std::ostream& out, Ring5 const& r) {
    out << "Ring5{"<<r.value()<<'}';
    return out;
}
```

We obtain the inequality operator as shown for class `Date` through inheriting from `boost::equality_comparable<Ring5>`[DAFo8].

Implementing our operations for addition and multiplication can either be done through implementing the combined assignment operators within the class and then implement the binary operators as non-member functions relying on those. Or by letting the Boost.Operators library provide us with the non-member functions that are implemented schematically. I'll show you both variants. First the multiplication operation in a DIY fashion.

```
struct Ring5
{
//...
    Ring5 operator*=(Ring5 const&r) {
        val = (val * r.value())%5;
        return *this;
    }
};
inline Ring5 operator*(Ring5 l, Ring5 const &r){
     l *= r;
     return l;
}
```

Note that `operator*` is implemented in the header file, because it is so short. To avoid violating the *one-definition-rule*, we need to mark this function definition with the keyword **`inline`**. If a member function is defined within a class, it is implicitly marked as an *inline function*. We also use a trick that saves us an additional local variable for computing the result. Instead of schematically passing both operands by const-reference, we use a pass-by-value parameter for the left-hand operand. This allows us to use its implicit local variable to perform the calculation.

inline function

Employing the Boost.Operators[DAF08] library allows us to save a bit of coding for the addition operator.

```
struct Ring5
: boost::equality_comparable<Ring5>
, boost::addable<Ring5>
{
//...
    Ring5 operator+=(Ring5 const &r) {
        val = (val + r.value())%5;
        return *this;
    }
};
```

That looks like we have sufficiently implemented the ring operations for our modulo 5 arithmetic type. However, using class `Ring5` is quite annoying. We can not easily combine it with integers. We always have to explicitly convert and integer value `x` first to a `Ring5 {x}` before it can take place in any operation. In addition, we can not easily assign a `Ring5` value to an integral variable as well, because there is no direct conversion, you have to use the member function `value()` instead explicitly.

Implicit Conversions

There are several ways to obtain more convenience. All of them have the hidden danger of ambiguities that make the solution harder to use for clients of our arithmetic type, especially when you provide more than one means. I'll show you them, so that you can make a

conscious decision in the future.

Let us again start with a simple test case. In addition to just add an integer literal it also checks that the resulting value is still of type `Ring5` by using the compile-time reflection available in C++11. The operator **`typeid`**`(type)` returns a reference to the type's `std::`**`type_info`** data structure provided by the compiler. Because we can not compare `std::`**`type_info`** objects directly, we rely on comparing the strings returned by its `name()` member function. To obtain the underlying type of a value, we can use the operator **`decltype`**`(value )`.[22]

typeid

type_info

decltype

[22] See later in chapter **??**.

```
void testAdditionWithInt(){
    Ring5 two{2};
    auto four=two+2u;
    ASSERT_EQUAL(Ring5{4},four);
    ASSERT_EQUAL(typeid(Ring5).name(),
                 typeid(decltype(four)).name());
}
```

I do not show you the necessary change to the class, because that would just mean to remove the keyword **`explicit`** in front of the constructor. This will allow the compiler to automatically insert the construction of a `Ring5` value from the integer 2. Then it can apply **`operator+`** that takes two arguments of type `Ring5`.

What is still missing, is the easy assignment back to a variable of integral type. We have the member function `value()` but C++ allows also automatic conversion from a user-defined type. This is done through a special syntax of operator function, a *conversion operator*. Instead of providing a return type in front of the **`operator`** keyword, you write the result type after **`operator`**.

conversion operator

```
void testAssignmentBackToInt(){
    Ring5 three{8};
    unsigned u3=three;
    ASSERT_EQUAL(3u,three);
}
```

```
struct Ring5
{
//...
    operator unsigned() const { return val; }
};
```

However, that sounds too easy. And it is. While we now can seamlessly assign a `Ring5` value to an **`unsigned`** variable, this conversion adds an ambiguity when we combine a `Ring5` value and an integer value in an addition or multiplication. Our previous test case no longer compiles because of that ambiguity. The compiler could either select the conversion operator and sum 2 unsigned values, or it can apply the non-explicit conversion constructor and use **`operator`**

+ defined for `Ring5`. We can back out of that ambiguity to go back and making either or both conversion options **`explicit`** again. This, however, requires us to adapt our test cases accordingly. To get some flexibility, we just make our constructor **`explicit`** again and thus can still use integer addition. Please see that to use the `Ring5` **`operator`**+ we have to explicitly convert the `2` to be added, otherwise the conversion operator would be apply.

```
void testAdditionWithIntExplicitCtor(){
    Ring5 two{2};
    auto four=two+Ring5{2u};
    ASSERT_EQUAL(Ring5{4},four);
    ASSERT_EQUAL(typeid(Ring5).name(),
                 typeid(decltype(four)).name());
}
void testAssignmentBackToIntExplicitCtor(){
    Ring5 three{8};
    unsigned u3=three;
    auto eight= three+5u;
    ASSERT_EQUAL(8u,eight);
    ASSERT_EQUAL(3u,u3);
    ASSERT_EQUAL(3u,three);
}
```

You might ask: "With all this ambiguity, where are non-explicit conversion operators useful?" I only recommend them for your own classes only in the case, where your class has a kind of valid versus invalid state. For example, you already encountered the case that a `std::istream` object can be invalid, *e.g.*, because of the end-of-file condition. You can employ this in conditions like **`while`**`(std::cin){...}`. Here an explicit conversion **`operator bool`**`()` actually is applied to obtain a condition from the stream object.

To sum up this section we can say that if you provide a conversion operator and a conversion constructor you better make both of them explicit, to avoid ambiguity traps for your class' users. Next, I show you another option to allow mixed arithmetic.

Mixed Arithmetic

After we have seen that automatic conversion can be a double-edged sword, I show you another option to provide mixed arithmetic. We can define overloads of our addition and multiplication operators that actually combine unsigned integers with our `Ring5` values resulting in `Ring5` values. While Boost.Operators[DAFo8] also provides means to get these mixed operators generated for you, I prefer to show you the DIY version, because with that it might be easier for you to understand the principles first and you can figure out how to employ the two-template argument version of the magic classes provided by Boost.Operators.

The key to allowing mixed arithmetic is to overload the opera-

tors combining the different types, *e.g.*, provide `operator+(Ring5, unsigned)` and `operator+(unsigned,Ring5)` additionally. We reuse the test cases from the previous section and I only show you the necessary extensions to the `Ring5.h` header file. I implement all operator functions inline for brevity and to show you all the different possibilities.

Most of the addition operators are implemented as const member functions if the left-hand operand is a `Ring5` value. However, you can not implement a binary operator as a member function, if the left-hand operator is of a different type than your class. This means we need to implement it as a regular function. To allow for an efficient implementation directly within the class context we define it as a **friend** function. In addition to the ability to define a foreign function inline within a class' definition, the keyword **friend** also provides access to all of the class' internals, *i.e.*, its **private**: and **protected**: members to the function declared/defined as friend within the class. I recommend to avoid friend functions or friend classes[23] if ever possible. Here I use the friend functions to show the concept and an area where using an inline friend function is actually tolerable. The base classes we used from Boost.Operators[DAFo8] actually work that way.

friend function

[23] Declaring a class as a friend is a shorthand for declaring all its member functions as friends.

The multiplication operator functions show further variations of the theme and options on how to implement the non-member functions without **friend**ship that you usually should prefer, if you can not or do not want to rely on Boost.Operators' mechanisms.

Compile-time calculation through `constexpr`

If you look closely at our different implementations of class `Ring5` you might see, there is still some difference in the way it behaves with respect to the built-in types, even though I promised we can make it behave as efficient and convenient to use. What is missing, is that while you can compute `3 * 4` at compile time, *e.g.*, when using it in a constant expression, there is no guarantee that `Ring5{3} * Ring5{4}` is computed at compile time. Either as an optimization, or for using it in a constant expression context (*e.g.*, `std::array< int, (Ring5{3}*Ring5{4} ).value() > a{};`).

Fortunately, our class `Ring5` and most of its computations are simple enough that we can implement them as **constexpr** functions and make our class a so-called *literal type*. This means, we modify the implementation so that its constructor gets the additional keyword **constexpr**.[24] Defining a function as **constexpr** is allowed for constructors making a class a literal type, member functions of literal types and free functions that only use literal types as local variables, parameters and return type. When used in compile-time computation the control flow must not throw an exception and must not have side effects outside of local variables and parameters. As a rule of thumb, anything that requires to allocate or refer to memory is verboten, for example, `std::string` or `std::vector`.

literal type

[24] Trivial classes and scalars (*e.g.*, `int`) and arrays of literal types are also literal types.

A constexpr constructor guarantees, that the compiler can evaluate the constructor at compile time, if it is called within a constant expression context. Also the `value()` member function and the conversion operator can be made **`constexpr`**. The easiest way to test that it is actually evaluated at compile time, is to use a **`static_assert`**.

static_assert

This assertion works like a unit test at compile time. If its evaluation fails, the compiler will show the error message string given as the second argument and won't compile your code successfully. For example, the static_assert declaration `static_assert(sizeof(int)>=8,"expect at least 64bit int");` anywhere in your program, checks if your compiler is treating type **`int`** as a 64bit value (assuming 8 bit **`char`**). You should use **`static_assert`** every time, you make specific assumptions about any compiler or code feature that is *implementation defined* to make your code future proof and not only document your assumptions, but also get notified, when your assumptions are violated.

We use it here, to check our **`constexpr`**ness of the constructor and functions we defined for a literal type version of class `Ring5`. If it compiles successfully the compiler evaluated the **`constexpr`** functions already.

```
static_assert(Ring5{2}==Ring5{4}+Ring5{3},
              "constexpr addition, equal and ctor");
static_assert(Ring5{}!=Ring5{4}*Ring5{3},
              "constexpr multiplication unequal");
static_assert(2u==(Ring5{4}*Ring5{3}).value(),
              "constexpr value()");
static_assert(2u==unsigned{Ring5{4}*Ring5{3}},
              "constexpr conversion");
```

A complete implementation without the mixed arithmetic and without using Boost.Operators, because the Boost library I am using, doesn't support C++'s **`constexpr`** yet. Please note, that the output **`operator<<`** can never be declared as **`constexpr`**, because it inherently has a side effect on a non-constexpr stream object which will never be able to happen at compile-time.

```
struct Ring5 {
    explicit constexpr
    Ring5(unsigned x=0u) : val{ x % 5 } {}
    constexpr unsigned value() const { return val; }
    constexpr operator unsigned() const { return val; }
    constexpr bool operator==(Ring5 const &r) const {
        return val == r.val;
    }
    constexpr bool operator!=(Ring5 const &r) const {
        return !(*this == r);
    }
    constexpr Ring5 operator+=(Ring5 const &r) {
        val = (val + r.value())%5;
        return *this;
    }
    constexpr Ring5 operator*=(Ring5 const&r) {
        val = (val * r.value())%5;
        return *this;
    }
    constexpr Ring5 operator+(Ring5 const &r) const {
        return Ring5{val+r.val};
    }
    constexpr Ring5 operator*(Ring5 const &r) const {
        return Ring5{val*r.val};
    }
private:
    unsigned val;
};
```

Simpler Constants of a User-defined Type

After we have made our class ready for compile-time evaluation, one difference to the built-in type remains. It is a bit unpleasant to create a value, because we always have to spell out the class name and braces, *e.g.*, `Ring5{1}`. Writing `1u` for creating the similar value of type **`unsigned`** is much more compact. And I have chosen quite a short name for our type, I could have made it even more unpleasant if I would have called it `RingIntegralNumbersModuloFive{1}`. You could have used a type alias

```
using Ring5=RingIntegralNumbersModuloFive;
```

to abbreviate that again, but haven't gained the pleasantness of `1u`.

user-defined literals

Fortunately for us, C++11 introduced a feature called *user-defined literals* (UDL). By defining special operator functions, we can provide a short-hand notation for literal constants of a user-defined type. I'll show you here only the most simple form sufficient for `Ring5` and will elaborate later on all the available forms (chapter **??**).

An operator for UDL defines a suffix, that if appended to a literal constant will make the compiler call that function. If the UDL operator is defined as **`constexpr`** that function application is done at

compile time.

```
namespace R5{
constexpr Ring5
operator"" _R5(unsigned long long v) {
    return Ring5{static_cast<unsigned>(v%5)};
}
}
```

The signature of the **operator**"" is predefined. The name given must be separated by a blank and it must begin with an underscore (_), because other names are reserved for future standardization. Since we are converting integer values, the parameter type is **unsigned long long**. That means, whenever we write something like `4_R5` it will be automatically converted to an instance of class `Ring5`.

To avoid a compiler warning and ensure the conversion will work fine, we first apply the modulo operator and then to adjust the type to the parameter type of `Ring5`'s constructor do a **static_cast <unsigned>**. This type cast is save, since we know that the value is between `0` and `4` already.

It is good practice to place UDL operators in a separate namespace and introduce them to the scope where you intend to use them via a **using namespace** directive or a **using** declaration. Otherwise you might end up with conflicting definitions of the same suffix. This is also one of the few cases where **using namespace** can be better than a using declaration, especially when several UDL operators are defined and to be used together. Without introducing the UDL operator into the scope where you use it, you'd loose the simple syntax of the suffix, but you can call the UDL operator function explicitly, if you like.

```
auto dont_call_this_way=R5::operator"" _R5(42);
```

Again, we can employ a **static_assert** to ensure the compile-time evaluation of the UDL-operator. Note also the using declaration of the UDL operator that omits the specific parameters.

```
using R5::operator"" _R5;// or using namespace R5;
static_assert(Ring5{2}==7_R5,"UDL operator");
```

I leave it to you as an exercise to beautify all unit tests to use the UDL suffix instead of the explicit construction of our `Ring5` values.

4.7 *Exercises*

Understand Move instead of Copy

Take the program tracer.cpp (Listing 4.1) and vary it, so that instead of defining a copy constructor define a move constructor. Within the move constructor change the parameter object's name to `name+ "moved from"` and define the name of the moved-to object to be the other's original name. How does the behavior of the output change

with respect to the variant with the copying tracer. You might need to wrap the local variable passed to function `foo` with `std::move` to have the program compile with the move-only class `Tracer`.

A value class for Words

string -> word, ignore punctuation and numbers on input, make it sortable by ignoring case

A class for simulating 7-Segment digits

A class for simulation 7-Segment Display

Subtraction and Division for Finite Field Modulo 5

As an intellectual exercise and also as a drill for operator overloading I'd like to ask you to change the class `Ring5` into a finite field `Field5`. In mathematics such a structure also provides the inverse operations to addition and multiplication. However, unlike these the inverse operations are less straight forward to implement. For example, to figure out the inverse for multiplication you need to consider the value you need to multiply with to get a remainder 1, which is the unit value of multiplication. For example, the multiplicative inverse of 2 is 3, because $3 \times 2 = 6\%5 = 1$, as a result $1/2 = 3$. Also the additional inverse of 2 is 3, because $2 + 3 = 5\%5 = 0$ and 0 is the unit value of addition. So $4 - 2 = 4 + 3 = 7\%5 = 2$ or $2 - 4 = 2 + 1 = 3$. You see, subtraction is not that problematic. For division, you should first elaborate all inverses on paper and write corresponding test cases to guarantee your implementation is correct. Note, there is no need and no possibility for a multiplicative inverse of zero.

Operations Table for Finite Field Modulo 5

refer to exercise multiplication table

vary: create a Ring6 class and provide its operation table, does every element have a multiplicative inverse?

Experiment wiht static_assert

Enumerations and Integers: `Month{may} vs.Month(5)`

Unfortunately there is no simple means to check if an integer is a valid value for an enumeration type. While an enumeration value can easily participate in integer arithmetic, you have to explicitly pry an integral value back into an enumeration type. And, there is no generic means to guarantee it will actually represent a valid enumeration value as it would be for conversion among integer types by using `std::numeric_limits<T>`. In the case of a bitfield enum type this might even be desired to store combined values in an enum variable that are not explicitly defined.
Initializing an enumeration variable or creating an enumeration value from an integer value actually means you as a programmer have to ensure that nothing bad happens. At least the syntax will show you that you have to transmogrify the integer into an enumeration. Instead of initializing with curly braces

```
Month nice{may};
```

you need to use parenthesis to make a `Month` value from an integer:

```
Month alsoMay=Month(5);
```

or even more obvious and lengthy use a **`static_cast`**:

```
Month pryToBeMay = static_cast<Month>(feb+3);
```

Some programmers use a sentinel enumeration value that represents the highest possible value and use that sentinel value to check against when assigning an integer. For example, code using the following enum can check if the to-be-assigned value is less or equal `max_color` to prevent running out-of-range, which can be eventually unpleasing when an enumeration variable is used to index a data structure without range checking.

```
enum class color {
    red, yellow, green,
    max_color=green
};
```

However, that practice requires one to update `max_color`'s definition, each time an enumeration value is inserted after `green` and also to program explicit checks, whenever an integer assignment takes place, *i.e.*, by always using a factory function for your enumeration type. If you really want to make sure, you might need to wrap your enumeration into a class type and use its constructors to actually check for valid values passed in, as we have done with our `Date` class before. More on wrapping enumeration types to ensure safe assignment can be found in [Heno2] and [Sako3], for example.

Caution 4.2: No Range Check for Enumerations

Listing 4.4: mixed/Ring5.h

```
#ifndef RING5_H_
#define RING5_H_
#include <iosfwd>
#include <boost/operators.hpp>
struct Ring5
: boost::equality_comparable<Ring5> {
    explicit Ring5(unsigned x=0u) : val{ x % 5 } {}
    unsigned value() const { return val; }
    explicit operator unsigned() const { return val; }
    bool operator==(Ring5 const &r) const {
        return val == r.val;
    }
    Ring5 operator+=(Ring5 const &r) {
        val = (val + r.value())%5;
        return *this;
    }
    Ring5 operator+=(unsigned r) {
        val = (val + r)%5;
        return *this;
    }
    Ring5 operator+(Ring5 const&r) const {
        Ring5 res{*this};
        res += r;
        return res;
    }
    friend Ring5 operator+(unsigned l, Ring5 const &r) {
        return Ring5{l+r.val};
    }
    friend Ring5 operator+(Ring5 l, unsigned r) {
        l.val = (l.val + r)%5;
        return l;
    }
    Ring5 operator*=(Ring5 const&r) {
        val = (val * r.value())%5;
        return *this;
    }
    Ring5 operator*=(unsigned r) {
        val = (val * r)%5;
        return *this;
    }
    Ring5 operator*(unsigned r) const {
        Ring5 res{r};
        res.val = (res.val * val)%5;
        return res;
    }
private:
    unsigned val;
};
inline Ring5 operator*(Ring5 l, Ring5 const &r){
     l *= r;
     return l;
}
inline Ring5 operator*(unsigned l, Ring5 const &r) {
    return Ring5{l*r.value()};
}
std::ostream& operator <<(std::ostream& out, const Ring5& r);
#endif /* RING5_H_ */
```