AdvancedC++: Operator Overloading and Template Meta-Programming

ROBERT JASKOLLA, MAXIMILIAN WARTNER

30.01.2018

Contents

Advanced C++: Operator Overloading

Operators

Definition: A operator is a symbol for the compiler to perform specific mathematical, logical manipulations or other special operations.

A binary operator takes two operands:

```
int i;
i = 7+7;           // i = 14;
```

A unary operator takes one operand:

```
int i = 0;        // i = 0
i += 7;           // i = 7;
```

Example:

- arithmetic operator: + , -, *, /
- logical operator: && and ||
- pointer operator: & and *
- memory management operator: new, delete[ ] ___ ## Operator Overloading
- arithmetic operator such as + are already overloaded in C/C++ for built-in types:

```
int result = 3 + 4; // result = 7
```

- different algorithms are used to compute different types of addition:

```
2 / 3 //Integer division: Result is 0
2.0 / 3.0 //Floating-point division: Result is 0.6666667
```

- different calls:

```
int main()
{

    int add(int left, int right)
    {
        return left + right;
    };

    int first = 43;
    int sec = 3465;

    int FirstResult = first + sec;
    int SecondResult = +(first + sec); //does not behave like expected
    int ThirdResult = add(first, sec); //see function above
}
```

The User can address the overloaded build-in definition of the +-operator for the basic int type in different ways. To simplyfy the programmcode and to have a method for overloading operators for custom types, Operator Overloading was implemented in C++.

Overloaded operators have appropriate meaning to user-de

fined types, so they can be used for these types. e.g. to use operator + for adding two objects of a user-de

fined class.

Overloading an Operator

```
class complex
{
    int real, imag;
public:
    void input()
    {
        //assign the ints..
    }
    ...

    complex operator+(complex A, complex B)
    {
        complex C;
        C.real = B.real + A.real;
        C.imag = B.imag + A.imag;
    }
    complex operator+(complex A, int i)
    {
        complex C;
        C.real = A.real + i;
        c.imag = A.imag;
        return C;
```

```cpp
    }
}

int main()
{
    complex c1, c2;
    c1.input();

    c2 = c1 + 7; // Compiler searching for overloaded function in class complex
}
```

When an operator appears in an expression, and at least one of its operands has a **class type or an enumeration type**, then overload resolution is used to determine the user-defined function to be called among all the functions whose signatures match it.

This allows to define the behavior of operators when applied to objects of a class. There is no reason to overload an operator except if it will make the code involving your class easier to write and especially easier to read!

Overloadable/Non-overloadable Operators

Following is the list of operators which can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	–
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded:

::	.*	.	?:	sizeof	typeid

Guidelines

- OO does not allow altering the meaning of operators when applied to built-in types, therefore one of the operands must be an object of a class
- OO does not allow defining new operator symbols, only those provided in the language can be used to override for a new type of data
- OO does not allow to change the number of operands an operator expects, precedence and associativity of operators or default arguments with operators
- OO should not change the meaning of the operator (+ does not mean subtraction!), the nature of the operator (3+4 == 4+3) or the data types and residual value expected
- OO should provide consistent definitions (if + is overloaded, then += should also be overloaded)

Syntax

```
T operator+(T A, T B)
{
    // do the overloading
    return T;
}
```

Overloading operators is similar to overloading functions, except the function name is replaced with the keyword operator with the operator's symbol added. Overloading operators for userdefined objects makes the code easier to understand and it is sensitive to the applications context. Therefore it is simple to understand the addition of the class apples or the addition of two fraction-objects.

Call the overloaded Operator:

```
object2 = object2.add(object1);          // use function
object2 = operator+(object2, object1);   // use OO like a function call
object2 = object2 + object1;             // use the simple operation
```

Types of Operators and canonical Implementations

Unary Operators

should always be overloaded as members, since the first argument must be an object of a class.

```
class complex
{
    int real, imag;
    complex operator+=(int i)
    {
        complex c;
        c.real = real + i;
        c.imag = imag;
        return c;
    }
}
```

Binary operators / shift operators

should always be overloaded as friend function, since often the new function may require access to private parts of the object.

```
class complex
{
    int real, imag;
public:
    friend complex operator+(complex A, complex B);
}
complex operator+(complex A, complex B)
{
    complex c;
    c.real = B.real + A.real;
    c.imag = B.imag + A.imag;
    return c;
}
```

Binary operators are typically implemented as non-members, to embody the possibility to add a integer to the complex object. (If the +-operator is only implemented as member function, only complex+integer would compile,integer+complex would result in a compile error.)

Conversion operator

The conversion operator is used for converting the object to another class (even base types).

```cpp
struct Fraction
{
    int numerator;
    int denominator;

    operator float() const { return numerator *1.0f / denominator;}
};
int main()
{
    Fraction fract;
    fract.numerator = 3; fract.denominator = 4;
    float value = fract;                        // value = 0.75
}
```

Canonical implementations

Commonly overloaded operators have the following typical, canonical forms:

Canonical copy-assignment operator

is expected to perform no action on self-assignment, and to return by reference.

```cpp
//assume the object holds reusable storage, such as a heap-allocated
    buffer mArray
T& operator=(const T& other) // copy assignment
{
    if(this != &other) {                // self-assignment check
        expected
        if(other.size != size) {            // storage cannot be
            reused
            delete[] mArray;                // destroy storage in this
            size = 0;
            mArray = nullptr;               // preserve invariants in
                                            // case next line throws
            mArray = new int[other.size];   // create storage in this
            size = other.size;
        }
        std::copy(other.mArray, other.mArray + other.size, mArray);
    }
    return *this;
}
```

Canonical move assignment

is expected to leave the moved-from object in valid state.

```
T& operator=(T&& other) noexcept // move assignment
{
    if(this != &other) { // no-op on self-move-assignment (delete[]/
       size=0 also ok)
        delete[] mArray;            // delete this storage
        mArray = std::exchange(other.mArray, nullptr); // leave moved-
           from in valid
                                                       // state
        size = std::exchange(other.size, 0);
    }
    return *this;
}
```

Stream extraction and insertion

These take a std::istream& or std::ostream& as the left hand argument. They are also known as insertion and extraction operators and have to be overloaded as non-members, due to the user-defined type as the right argument.

```
class complex
{
    int real, imag;
public:
    friend std::ostream& operator<<(std::ostream& stream, const complex
       & arg);
}
std::ostream& operator<<(std::ostream& stream, const complex& arg)
{
    cout << "{" << arg.real << "+" << arg.imag << "i" << "}";
    return stream;
}
```

Function call operator

When a user-defined class overloads the function call operator (operator()) it becomes a FunctionObject type. Standard algorithms (std::sort …) accept objects of such types to customize behavior.

```
struct Sum
{
    int sum;
    Sum() : sum(0) { }
    void operator()(int n) { sum += n; }
};
Sum s = std::for_each(v.begin(), v.end(), Sum());
```

Increment and decrement

The increment / decrement can be used in both the prefix and postfix form, but with different meanings:

In the prefix form, the residual value is the post incremented or post decremented value. In the postfix form, the residual value is the pre incremented or pre decremented value. These are unary operators, so they should be overloaded as members.

To distinguish the prefix from the postfix forms, the C++ standard has added an unused argument (int) to represent the postfix signature. Since these operators should modify the current object,they should not be const members.

```
struct X
{
    X& operator++()
    {
        //actual increment takes place here
        return *this;
    }
    X operator++(int)
    {
        X tmp(*this);   // copy
        operator++();   // pre-increment
        return tmp;     // return old value
    }
};
```

Relational operators

Standard algorithms such as std::sort can be used for user-provided types, if the operator< is overloaded.

```cpp
struct Record
{
    std::string name;
    unsigned int floor;
    double weight;
    friend bool operator<(const Record& l, const Record& r)
    {
        return std::tie(l.name, l.floor, l.weight)
            <  std::tie(l.name, l.floor, l.weight); // keep the same order
    }
}
```

Typically, once operator< is provided, the other relational operators are implemented in terms of operator<:

```cpp
inline bool operator< (const X& lhs, const X& rhs){ /*do the actual comparison*/ }
inline bool operator> (const X& lhs, const X& rhs){ return rhs < lhs; }
inline bool operator<=(const X& lhs, const X& rhs){ return !(lhs > rhs); }
inline bool operator>=(const X& lhs, const X& rhs){ return !(lhs < rhs); }
```

Likewise, the inequality operator is typically implemented in terms of operator==:

```cpp
inline bool operator==(const X& lhs, const X& rhs){ /*do the actual comparison*/ }
inline bool operator!=(const X& lhs, const X& rhs){ return !(lhs == rhs); }
```

Bitwise arithmetic operators

User-defined classes and enumerations that implement the requirements of BitmaskType are required to overload the bitwise arithmetic operators:

operator&	operator|	operator^	operator&=	operator|=	operator~

and may optionally overload the shift operators:

operator<<	operator>>	operator>>=	operator<<=

The canonical implementations usually follow the pattern for binary arithmetic operators described above.

Rarely overloaded operators

- The address-of operator : operator&
  - there are little to no use-cases that need to overload the default behavior of the address operator
- The boolean logic operators : operator&& and operator||
  - Unlike the built-in versions, the overloads cannot implement short-circuit evaluation
  - the overloads do not sequence their left operand before the right one
- The comma operator : operator,
  - Unlike the built-in version, the overloads do not sequence their left operand before the right one
- The member access through pointer to member : operator->*
  - no specific downsides to overloading this operator, but it is rarely used in practice

Best Practises

- there is no reason to overload an operator, if it won´t make the code easier to understand

```cpp
// The following lines are hard to understand:
// Skalarproduct returns int
friend int operator*(const Vector3d& left, const Vector3d& right);
// Crossproduct returns new Vector
friend Vector3d operator*(const Vector3d& left, const Vector3d& right);

// much better:
friend int SkalarProduct(const Vector3d& left, const Vector3d& right);
friend Vector3d CrossProduct(const Vector3d& left, const Vector3d&
    right);
```

- Overloading an operator should always be implementet as native as possible, therefore when adding fractions there should be a fraction returned and not an int

```cpp
// Bad:
```

```cpp
int operator+(Fraction A, Fraction B)
{
    //...
    return 10;
}

// Good:
Fraction operator+(Fraction A, Fraction B)
{
    Vector result;
    //...
    return result;
}
```

- Overloading should be defined in the library namespace, because there is no need to pollute the global namespace and additionally the syntax will be less verbose

```cpp
namespace Lib {
    class A { };

    A operator+(const A&, const A&);
} // namespace Lib
```

Conclusion

On one hand operator overloading makes your program easier to write and to understand, on the other hand overloading does not actually add any capabilities to C++. Everything you can do with an overloaded operator you can also do with a function. However, overloaded operators make your programs easier to write, read, and maintain which in fact is a great benefit! I would personally recommend to overload operators for defined objects, if it fits the application.

Template Meta Programming

Templates in C++

Templates offer the possibility in C++ to program in a generic way and to create type-safe containers. There are class templates and function templates. The following example shows a function template. It should be noted here that the keyword typename can also be replaced by class. For simple templates, there is no right or wrong. In the main method, the template is used in different ways. initially it is applied to the types int and double. in the next line you see a special case, different types are compared. this is not possible here, but with the addition the integer is cast on double.

```cpp
#include <iostream>

template <typename T>
T max (const T& first, const T& second)
{
    return (first >= second) ? (first) : (second);
};

int main()
{
    std::cout << max(10, 20) << st::endl;
    std::cout << max(8.43, 23.22) << std::endl;
    std::cout << max<double>(13.44, 10 << std::endl;
    return 0;
}
```

History of Template Meta Programming

Templates are turing-complete. Also, they were evaluated at the compile time. That means, that all functions, that can be calculated, can be calculated with C++ Templates at compile time. In 1994, Erwin Unruh von Siemens-Nixdorf presented a program to the C++ Standard Committee, which calculated the primes up to 30 and returned them in the form of error messages. Thus he proved this fact.

Basic Techniques of Template Meta Programming

The following section will first provide an overview of important basics of template meta-programming.

Functions

Template metaprogramming requires functions that produce a result already at the compile time. But that's about classic C ++ Function definitions hardly possible. Therefore, one uses a trick: With the function arguments a class template is parameterized. The result of the function can be statically extracted via a constant member.

```
template <unsigned int x, unsigned int y>
struct add
{
    enum { value = x + y };
};
```

It should also be noted that the parameters are not limited to the data type unsigned int. all discreet data types can be used.

Enumeration vs. constant variable

Again and again it is discussed whether the presentation of results in template functions over a one-element enumeration or maybe better over a constant one Class variable should happen:

```
template <int x>
struct id_enum
{
    enum { value = x };
};

template <int x>
struct id_static
{
    static const int value = x;
};
```

Parameterizing a class template that contains class variables causes the compiler to create and instantiate them in static memory. Values of enumerations are not lvalues, so have no address and are treated as literals by the compiler. Since, according to the definition of the template metaprogramming, this refers exclusively to computation or processing at compile time, and the effect of class variables relates to translation time, and the effect of class variables goes beyond compile time, enumerations are preferable.

Recursion

In TMP there are no variable variables, only constant ones. Therefore there can be no loops. Instead of these, in TMP recursions are used. A recursion cancellation is due to the specialization of the template for the corresponding Case defined.

The following code shows a template for calculating the faculty of an integer. The first template defines the recursion step n * (n - 1) !. It instantiates itself, as long as there is a decrease in n by 1, until the case n = 0, which is given here by the specialized template with factorial <0>, occurs. It should also be noted that in the specialization factorial <0> the specified parameter 0 in angle brackets is directly behind the template name and the angle brackets behind the keyword "template" remain empty. If the template has several parameters, the unspecified ones remain in the upper bracket while the specified ones are written to the lower one.

```
template <unsigned int n>
struct factorial
{
    enum { value = n * factorial <n - 1 >:: value };
};

template <>
struct factorial <0>
{
    enum { value = 1 };
};
```

Type Functions

Type functions generally mean functions that use a data type instead of a value as return value or make their result dependent on a data type.

The following example shows the template number_type, which returns a corresponding data type depending on how many bits are passed. You can also see the template bitsize, which returns the number of bits of a data type, and the template bigger_type, which retrieves and returns the next largest data type from the previous two templates.

```
#import <iostream>

template <int bits>
struct number_type
{
    typedef int type;
};
template <>
struct number_type <16>
{
    typedef short type;
};
template <>
struct number_type <8>
{
    typedef char type;
};

template <typename arg >
struct bitsize
{
    enum { value = sizeof ( arg ) * 8 };
};

template <typename arg >
struct bigger_type
{
    typedef typename number_type < bitsize <arg>:: value * 2 >::type
        type;
};

int main()
{
    std::cout << typeid(number_type<16>::type).name() << std::endl;
    std::cout << typeid(short).name() << std::endl;
    std::cout << typeid(number_type<64>::type).name() << std::endl;
    std::cout << typeid(number_type<8>::type).name() << std::endl;
    std::cout << bitsize<number_type<16>::type>::value <<std::endl;
    std::cout << typeid(bigger_type<number_type<8>::type>::type).name()
         << std::endl;
    return 0;
}
```

Recursion and conditional branching

A conditional branch is a branch in the program flow by, for example, an if-else construct or the ternary expression. The evaluation of the condition happens here only at runtime. As already mentioned, however, such post-compilation time effects in the Template Meta Programming should be avoided. Actually, there are several ways to evaluate conditions already at compile time and depending on them to branch out.

The following example shows three templates which should calculate, whether an integer is a prime number or not. First, the template IfThenElse can be seen. This is a self-built branched condition. The first parameter is a boolean, second and third parameters are integers, which are returned if the boolean is either true or false.

In the template is_prim_check the IfThenElse template will be used. If a number n is divisible by i, then 0 is returned, otherwise is_prim_check <i-1, n> is called. Termination condition of this recursion is the specialization is_prim_check <1, n>, in this case 1 is returned. The template is_prim checks whether the transferred integer n is a prime number or not and returns the value 0 or 1 accordingly. If 0 or 1 was passed as n, 0 will be returned, since these are not prime numbers. Otherwise the template is_prim_check with the parameters n / 2 and n is called and returned.

```
template <bool cond , int true_part , int false_part>
struct IfThenElse;

template <int true_part , int false_part>
struct IfThenElse <true , true_part , false_part>
{
    enum { value = true_part };
};

template <int true_part , int false_part>
struct IfThenElse <false , true_part , false_part>
{
    enum { value = false_part };
};


template <int i, int n>
struct is_prim_check
{
    enum { value = IfThenElse < ((n % i) != 0), ( is_prim_check <i - 1,
        n >:: value ),
           (0) >:: value };
};

template <int n>
struct is_prim_check <1, n>
{
    enum { value = 1 };
};


template <int n>
```

```
struct is_prim
{
    enum { value = is_prim_check <(n / 2), n >:: value };
};

template <>
struct is_prim <1>
{
    enum { value = 0 };
};

template <>
struct is_prim <0>
{
    enum { value = 0 };
};
```

Advanced Concepts

Unrolled loops

Quite often happens that for certain calculations, in particular at the use of dynamic data structures, loops are used. In time-critical applications, loops are often "rolled up", if possible. Rolling up means that the instructions are repeated within a loop written among each other, so that can be dispensed with the loop. This can greatly reduce the number of total instructions executed. Since all control instructions of the loop are eliminated. In addition, there are no jumps in pure calculations, which additionally increases the execution speed. A downside to manually rolled loops is that they add dynamics and portability of the code very restrict. For example, high rendition of algorithms would result for different configurations. Through template metaprogramming, loops can be modeled that have a generality and a dynamic character when used at development time to have. After compiling such code, the compiler then generates loopless, in the optimal case, minimal instruction sequences.

Expression templates

Expression templates are probably the prime example of template metaprogramming. Generally, templates are called expression templates, if they can manipulate specific expressions or even enable them. Let it be even using Expression Templates to model completely new languages, which can be used within C ++. One speaks here of so-called DSLs (domain specific languages) or DSELs (domain specific embedded languages). These are languages that usually have a high degree of abstraction and a high significance in terms of concrete problem areas.

Variadic templates

- Variadic templates take a variable number of arguments

```
template<typename... Values> class tuple; // takes zero or more
    arguments
```

- may also apply to functions

```
template<typename... Params> void printf(const std::string &str_format,
     Parmas... parameters);
```

- And use the ellipsis operator: (...), which declares a parameter pack.
- The use of variadic templates is often recursive

```
// recursive
template<typename T, typename... Args>
void printf(const char *s, T value, Args... args)
{
    while (*s) {
        if(*s == '%'){
            ++s;
        }
        else {
            std::cout << value;
            s += 2; // only works on 2-character format strings (%d, %f
                ...)
            printf(s, args...); // called even when *s is 0 but does
                nothing
            return;             //in that case
        }
    }
    std::cout << *s++;
}
// variadic template version of printf calls itself, or (in the event
    that args... is empty) calls the base case.
```

The variadic parameters themselves are not easy to access in the implementation of a function or class.

There is no simple mechanism to iterate over the values of the variadic template. In fact it needs to be implemented in one of several ways: * function overloading * using a dumb expansion marker

Conclusion

In my opinion, template meta programming is an interesting type of programming that you should always keep in mind. For special problems and especially for very time-critical applications, template meta programming can be a good choice. However, this is always associated with increased effort in programming and less readable and poorly maintainable code. therefore template meta programming should be used with caution and attention should be paid to how familiar the rest of the team is with the topic.