Tech/Doc

C++ : Bases : Operator Overloading Prototypes

You'll find below prototypes of default, common operator overloading, ready to be copy-pasted on your code.

Preamble: Class Methods vs. Functions?

In Object C++ (see , as well as Item 3), it is preferable to "prefer non-member non-friend functions to member functions.

So if you have the choice between two equal implementations, use the function instead of a class method.

Sources:
"*Item 23 - Prefer non-member non-friend functions to member functions*", in Effective C++ 3rd Edition by **Scott Meyers**
"*Item 32 - Name Lookup and the Interface Principle - Part 2*" in Exceptional C++, by **Herb Sutter**
"*Item 38 - Monoliths "Unstrung", part 2: Refactoring std::string*" in Exceptional C++ Style, by **Herb Sutter**
"*Monoliths "Unstrung"*", Guru of the Week #84 (http://www.gotw.ca/gotw/084.htm), by **Herb Sutter**

Arithmetic Operators

The arithmetic operators are usually used on concrete types. For exemple, matrices or complex classes. Iterators make uses of some of those operators (for example, `++` or `+=`).

Binary `+`, `-`, `*`, `/`, `%`

Despite being used on classes, having functions-type prototypes instead of methods is more extensible.

```
T operator + (const T & p_oLeft, const T & p_oRight) // A + B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}

T operator - (const T & p_oLeft, const T & p_oRight) // A - B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}

T operator * (const T & p_oLeft, const T & p_oRight) // A * B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}

T operator / (const T & p_oLeft, const T & p_oRight) // A / B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}

T operator % (const T & p_oLeft, const T & p_oRight) // A % B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}
```

Unary `++`, `--`

These operators can be class methods or functions.

These operators are tricky, so the following rules must be reminded:

- If you need to increment i, the code is: `++i`
- If you need to increment i and then retrieve its previous value, the code is: `j = i++`
- Never ever write `i++` alone (be it in a `for` instruction, etc.).
- The rules above deal with `--i` and `i--`, too.

```
T & T::operator++() // ++A
{
   // Do increment of "this" value
   return *this ;
}

T T::operator++(int) // A++
{
   T oCopy = *this ;
   // Do increment of "this" value
   return oCopy ;
}

T & T::operator--() // --A
{
   // Do decrement of "this" value
   return *this ;
}

T T::operator--(int) // A--
{
   T oCopy = *this ;
   // Do decrement of "this" value
   return oCopy ;
}
```

```
T & operator++(T & p_oRight) // ++A
{
   // Do increment of p_oRight value
   return p_oRight ;
}

T operator++(T & p_oRight, int) // A++
{
   T oCopy ;
   // Copy p_oRight into oCopy
   // Do increment of p_oRigh value
   return oCopy ;
}

T & operator--(T & p_oRight) // --A
{
   // Do decrement of p_oRight value
   return p_oRight ;
}

T operator--(T & p_oRight, int) // A--
{
   T oCopy ;
   // Copy p_oRight into oCopy
   // Do decrement of p_oRigh value
   return oCopy ;
}
```

Unary `+`, `-`

These operators are rarely used. Still, it makes senses to define them for concrete types aiming to behave like integer or real numbers, as it is always legal to write `int b = +a ;` or `int b = -a ;`.

```
T operator + (const T & p_oRight) // +A
{
   T oTemp ;
   // Do something about p_oRight, and put it in oTemp
   return oTemp ;
}

T operator - (const T & p_oRight) // -A
{
   T oTemp ;
   // Do something about p_oRight, and put it in oTemp
   return oTemp ;
}
```

Note that if you know the unary `+` operator gives the same value as the one given as a parameter, you could write it like, but you need to know what you're doing:

```
const T & operator + (const T & p_oRight) // +A
{
   return p_oRight ;
}

T & operator + (T & p_oRight) // +A
{
   return p_oRight ;
}
```

Assignement operator `=`

Note that the `operator =` is **always** a class method.

```
T & T::operator = (const T & p_oRight) // A = B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}
```

Assignement operators `+=`, `-=`, `*=`, `/=`, `%=`

But these assignement operators can be either methods or functions.

```
T & T::operator += (const T & p_oRight) // A += B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}

T & T::operator -= (const T & p_oRight) // A -= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}

T & T::operator *= (const T & p_oRight) // A *= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}

T & T::operator /= (const T & p_oRight) // A /= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}

T & T::operator %= (const T & p_oRight) // A %= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}
```

```
T & operator += (T & p_oLeft, const T & p_oRight) // A += B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}

T & operator -= (T & p_oLeft, const T & p_oRight) // A -= B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}

T & operator *= (T & p_oLeft, const T & p_oRight) // A *= B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}

T & operator /= (T & p_oLeft, const T & p_oRight) // A /= B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}

T & operator %= (T & p_oLeft, const T & p_oRight) // A /= B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}
```

Bitwise operators

These operators are very specific to bitwise manipulation. An example of use of those operators would be to simulate a bit container object, which will need to supply those operators to work as the bit-wise C version.

Shift and Assignement Shift operators `<<`, `>>`

Note that the `int` parameter can be of any type chosen by the user. It's just that semantically, the p_oRight parameter is supposed to be some kind of distance to "shift" the value.

Note, too, that the shift-operators used on C++ streams are not to be considered as arithmetics operators, and are described after.

```
T operator << (const T & p_oLeft, int p_iShift) // A = B << 3
{
   T oResult ;
   // Put into oResult the result of the shift
   return oResult ;
}

T operator >> (const T & p_oLeft, int p_iShift) // A = B >> 3
{
   T oResult ;
   // Put into oResult the result of the shift
   return oResult ;
}
```

Assignment-Shift operators `<<=`, `>>=`

The Shift operators section information can be applied here.

```
T & operator <<= (T & p_oLeft, int p_iShift) // A <<= 3
{
   // Apply the shift to p_oLeft
   p_oLeft.m_iValue00 <<= p_iShift ;
   return p_oLeft ;
}

T & operator >>= (T & p_oLeft, int p_iShift) // A >>= 3
{
   // Apply the shift to p_oLeft
   return p_oLeft ;
}
```

Unary "one's complement" `~`

The unary "one's complement" is not supposed to modify its operand.

```
T operator ~ (const T & p_oLeft) // ~A
{
   T oResult ;
   // assign the "one's complement" of p_oLeft to oResult
   return oResult ;
}
```

Bitwise binary `&`, `|`, `^`

Like the arithmetic binary operators `+`, `*`, etc., these operators are better coded as functions:

```
T operator & (const T & p_oLeft, const T & p_oRight) // A & B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}

T operator | (const T & p_oLeft, const T & p_oRight) // A | B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}

T operator ^ (const T & p_oLeft, const T & p_oRight) // A ^ B
{
   T oResult ;
   // Do the operation using oResult as result
   return oResult ;
}
```

Bitwise binary `&=`, `|=`, `^=`

Like the arithmetic binary operators `+=`, `*=`, etc., these operators can be coded as methods or functions, as convenient:

```
T & T::operator &= (const T & p_oRight) // A &= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}

T & T::operator |= (const T & p_oRight) // A |= B
{
```

```
T & operator &= (T & p_oLeft, const T & p_oRight) // A &= B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}

T & operator |= (T & p_oLeft, const T & p_oRight) // A |= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}

T & T::operator ^= (const T & p_oRight) // A ^= B
{
   // Do the assignement... Beware of auto assignement

   return *this ;
}
```

```
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}

T & operator ^= (T & p_oLeft, const T & p_oRight) // A ^= B
{
   // Beware of the case p_oLeft and p_oRight are the same object

   return p_oLeft ;
}
```

Comparison operators

These operators should return a `bool`.

Unary `!`

```
bool operator ! (const T & p_oRight) // !A
{
   bool bResult ;
   // Put the result of the operation applied to p_oRight
   return bResult ;
}
```

Binary `<`, `==`

These methods are usually the core of the comparisons. All other comparisons can be deduced through a combination of those operators with the operator `!`.

```
bool operator == (const T & p_oLeft, const T & p_oRight) // A == B
{
   bool bResult ;
   // Put the result of the operation applied to p_oLeft and p_oRight
   return bResult ;
}

bool operator < (const T & p_oLeft, const T & p_oRight) // A < B
{
   bool bResult ;
   // Put the result of the operation applied to p_oLeft and p_oRight
   return bResult ;
}
```

```
bool T::operator == (const T & p_oRight) // A == B
{
   bool bResult ;
   // Put the result of the operation applied to "this" and p_oRight
   return bResult ;
}

bool T::operator < (const T & p_oRight) // A < B
{
   bool bResult ;
   // Put the result of the operation applied to "this" and p_oRight
   return bResult ;
}
```

Binary `!=`, `>`, `<=`, `>=`

These functions can be deduced from the `==` and `<` operators. Thus, you can either import an utility namespace from the STL into the namespace of your class, which will do the work for you:

```
#include <utility>

namespace MyNamespace
{

using namespace std::rel_ops ;

class T
{
   // etc.
} ;


inline bool operator == (const T & p_oLeft, const T & p_oRight)
{
   // Etc.
}

inline bool operator < (const T & p_oLeft, const T & p_oRight)
{
   // Etc.
}

// as == and < are defined, the !=, >, <=, and >=
// operators will be automatically generated
// for type T if needed

} //  namespace MyNamespace
```

Another solution is defining them yourself:

```
bool operator != (const T & p_oLeft, const T & p_oRight) // A != B
{
   bool bResult ;
   // Put the result of the operation applied to p_oLeft and p_oRight
   return bResult ;
}

bool operator > (const T & p_oLeft, const T & p_oRight) // A > B
{
   bool bResult ;
   // Put the result of the operation applied to p_oLeft and p_oRight
   return bResult ;
}

bool operator <= (const T & p_oLeft, const T & p_oRight) // A <= B
{
   bool bResult ;
   // Put the result of the operation applied to p_oLeft and p_oRight
   return bResult ;
}

bool operator >= (const T & p_oLeft, const T & p_oRight) // A => B
{
   bool bResult ;
   // Put the result of the operation applied to p_oLeft and p_oRight
   return bResult ;
}
```

```
bool T::operator != (const T & p_oRight) // A != B
{
   bool bResult ;
   // Put the result of the operation applied to "this" and p_oRight
   return bResult ;
}

bool T::operator > (const T & p_oRight) // A > B
{
   bool bResult ;
   // Put the result of the operation applied to "this" and p_oRight
   return bResult ;
}

bool T::operator <= (const T & p_oRight) // A <= B
{
   bool bResult ;
   // Put the result of the operation applied to "this" and p_oRight
   return bResult ;
}

bool T::operator >= (const T & p_oRight) // A => B
{
   bool bResult ;
   // Put the result of the operation applied to "this" and p_oRight
   return bResult ;
}
```

Binay `&&`, `||`

Never ever overload those operators.

Construction/Destruction operators

In C, there is the stack, and there is the heap.

In C++, it gets somewhat more complicated: On top of that, there is the free store.

Why this distinction, as we know `new` will call, in the end, `malloc`?

Because we are wrong: `new` is not supposed to call `malloc`. Sometimes, it will. Sometimes, it won't. And this behaviour can even change for each class, as in C++ `new` (and `delete`) can be overloaded, for example, to use memory allocated from a pool, which can be either allocated on the heap, or even on the stack.

Overloading the `new`/`delete` operators should always be done for a very good reason.

`new`, `delete`

`new[]`, `delete[]`

Miscellaneous operators

Function Call `()`

The "function call" operator is usually used for Functors, that is, objects that are supposed to be able to behave like functions.

This can be confusing at first sight, but this is a very important feature when considering having generic code working on both old C way and the C++ way.

The "function call" has no set prototype, as it is supposed to work like any other function with the same prototype. It is more easily understood when compared to a regular method. Let's say we want to implement both a "function call" operator and an "doExecute" method with the same parameters:

```
short T::operator () (const std::string & p_strValue, bool p_bValue)
{
   // Etc.
}

short T::doExecute(const std::string & p_strValue, bool p_bValue)
{
   // Etc.
}

// etc.

void doSomethingWithT(T & p_oMyT)
{
   short s0 = p_oMyT.doExecute("Hello World", true) ;
   short s1 = p_oMyT("Hello World", true) ;
}
```

So, the only difference between the two method is that, in the prototype, we replace `doExecute` by `operator ()`, and that when using the method, we remove the `.doExecute` text altogether, and use the object is if it was a function.

Functors are usually encountered the first time when using `std::map`, `std::sort`, `std::foreach` or other APIs from the STL.

Array Subscript `[]`

The subscript works like the function-call with but two differences. The first is that it uses the `[]` instead of the `()`. The second is that only one parameter is authorized. It is used mostly to enable objects like `std::vector` or `std::string` to be accessed like their C array versions.

We can use almost the same prototype as above to describe the subscript:

```
short T::operator [] (const std::string & p_strValue)
{
   // Etc.
}

short T::doExecute(const std::string & p_strValue)
{
   // Etc.
}

// etc.

void doSomethingWithT(T & p_oMyT)
{
   short s0 = p_oMyT.doExecute("Hello World") ;
   short s1 = p_oMyT["Hello World"] ;
}
```

Note that in most cases, the subscript is seen and used as a kind of direct accessor through the reference of the accessed element (as it's supposed to work with C arrays). Thus, it should be accessible at least as a const-reference, or both const and non-const references:

```
const double & T::operator[](int p_iIndex) const // const double & d = A[25]
{
   return /* the const reference double */ ;
}

double & T::operator[](int p_iIndex)  // double & d = A[25]
{
   return /* the reference double */ ;
}
```

Indirection/Dereference `*`

Usually this operator will be used in C++ when create a "smart pointer" class and giving it pointer-like behaviours, in this case, the dereferencing.

This overload can be either a class method or a function.

```
class T ;
class TPtr ; // class simulating a (smart) pointer on T

T & TPtr::operator * () const // T * pT = *oPtr ;
{
   return /* some reference of T */ ;
}
```

```
class T ;
class TPtr ; // class simulating a (smart) pointer on T

T & operator * (const TPtr & p_oTPtr) // T * pT = *oPtr ;
{
   return /* some reference of T */ ;
}
```

Address-of/Reference `&`

This method is the anti-twin of the Dereference operator. Usually, when you want to wrap an object around another object, the wrapper must still behave like the original object, and thus, return its address if asked.

This overload can be either a class method or a function.

```
class T ;
class TPtr ; // class simulating a (smart) pointer on T

T * TPtr::operator & () const // T * pT = *oPtr ;
{
   return /* some address of T */ ;
}
```

```
class T ;
class TPtr ; // class simulating a (smart) pointer on T

T * operator & (const TPtr & p_oTPtr) // T * pT = *oPtr ;
{
   return /* some address of T */ ;
}
```

Member-by-Pointer `a->b`

Again, in the case of smart pointer utility class, simulating the `->` operator is necessary.

This overload must be a class method.

```
class T ;
class TPtr ; // class simulating a (smart) pointer on T

T * TPtr::operator -> () const
{
   return /* the pointer to T */ ;
}
```

Member by Pointer Function Pointer Indirection `a->*b`, `-`

Comma `a,b`

Do not overload it.

Cast

The cast operators are used to give the compiler a way to static_cast an object into another type. Avoid as much as possible the cast operators, as the compiler will use them if needed, potentially causing confusion as what the code is supposed to do, and what the code is really doing.

Note that the cast operator declaration declares no return type, which is no surprising because the return type is already in the name of the cast operator...

```
// static_cast<const char *>(ConstA)

T::operator const char * () const
{
   return /* something as a "const char *" */ ;
}

// static_cast<const char *>(A)
// static_cast<char *>(A)

T::operator char * ()
{
   return /* something as a "char *" */ ;
}
```

Last but not least, cast operators do not always behave as supposed at first glance, as shown by the code below:

```
void doSomething()
{
   T oAAA ;
   const T oBBB ;

   // call "operator char *" and NOT "operator const char *"
   // because oAAA is NOT const
   const char * p0 = static_cast<const char *>(oAAA) ;

   // call "operator char *"
   char * p1 = static_cast<char *>(oAAA) ;

   // call "operator const char *"
   // because oBBB is const
   const char * p2 = static_cast<const char *>(oBBB) ;
}
```

C++ Streams operators

The C++ streams use their own version of commun overloadable operators, different in their semantic signification than their arithmetic versions.

Insert/Extract operators `<<`, `>>`

These are the generic prototypes for data extracting from or inserting to a C++ stream.

```
T & operator << (T & p_oOutputStream, const CMyObject & p_oMyObject) // A << oMyObject
{
   // do the insertion of p_oMyObject
   return p_oOutputStream ;
}

T & operator >> (T & p_oInputStream, CMyObject & p_oMyObject) // A >> oMyObject
{
   // do the extraction of p_oMyObject
   return p_oInputStream ;
}
```

For your convenience, you'll find below a ready to copy/paste version for the C++ template streams:

```
// OUTPUT << A
template <typename charT, typename traits>
std::basic_ostream<charT,traits> & operator << (std::basic_ostream<charT,traits> & p_oOutputStream, const CMyObject & p_oMyObject)
{
   // do the insertion of p_oMyObject
   return p_oOutputStream ;
}

// INPUT >> A
template <typename charT, typename traits>
std::basic_istream<charT,traits> & operator >> (std::basic_istream<charT,traits> & p_oInputStream, CMyObject & p_oMyObject)
{
   // do the extract of p_oMyObject
   return p_oInputStream ;
}
```

Or the same version, limited to `char` streams:

```
// OUTPUT << A
std::ostream & operator << (std::ostream & p_oOutputStream, const CMyObject & p_oMyObject)
{
   // do the insertion of p_oMyObject
   return p_oOutputStream ;
}

// INPUT >> A
std::istream & operator >> (std::istream & p_oInputStream, CMyObject & p_oMyObject)
{
   // do the extract of p_oMyObject
   return p_oInputStream ;
}
```

And for the `wchar_t` streams:

```
// OUTPUT << A
std::wostream & operator << (std::wostream & p_oOutputStream, const CMyObject & p_oMyObject)
{
   // do the insertion of p_oMyObject
   return p_oOutputStream ;
}

// INPUT >> A
std::wistream & operator >> (std::wistream & p_oInputStream, CMyObject & p_oMyObject)
{
   // do the extract of p_oMyObject
   return p_oInputStream ;
}
```