Operator overloading

The following operator symbols can be defined by user:

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

The following can *not* be user defined:

```
::        .         .*        ?:
```

The following must be defined as *non-static member functions*:

```
=      [ ]       ( )       ->
```

This will guarantee that the left hand side operand will be an *lvalue* of the type in question.

Operator == for String

As *member*:

```
class String
{
public:
   …
   bool operator==(const String& rhs);
   …
};
```

As *non-member* (possibly as *friend*, if necessary):

```
class String
{
public:
   …
};

bool operator==(const String& lhs, const String& rhs);
```


Member or not?

```
s1 == s2
```

A binary member operator function have the left argument bound to **this**.

- written as an ordinary member function:

```
s1.operator==(s2)
```

- the left hand side operand must be a `String` object – **this** point to `s1`

A binary non-member function has both arguments as explicit parameters/arguments.

- written as an ordinary function call:

```
operator==(s1, s2)
```

- a non-explicit type converting constructor allows the left operand to be **const char**[] (**const char***):

```
"C++11" == s2
```

  – a temporary object is in such case created, as explicitly done below:

```
String{ "C++11" } == s2
```

  – if the constructor is **explicit** you must do it this way – explicit type conversion

Guidelines for deciding on, if an operator function should be a member or not will follow!

Friend or not?

A non-member operator function can be a **friend**.

```
class String
{
   public:
      …
      friend bool operator==(const String& lhs, const String& rhs);
      …
};

bool operator==(const String& lhs, const String& rhs);
```

Avoid, if possible (if there are public member functions that can be used to implement).

```
bool operator==(const String& lhs, const String& rhs)
{
   return strcmp(lhs.c_str(), rhs.c_str()) == 0;
}
```

Optimized versions – to avoid implicit type conversion and temporaries

There can be optimized versions for equality test with **char***.

```
bool operator==(const String& lhs, const String& rhs);

bool operator==(const String& lhs, const char* rhs);

bool operator==(const char* lhs, const String& rhs);
```

This allows the following equality tests, without any temporary objects being created:

```
String s1{ "foo" };
String s2{ "fie" };

char c3[]{ "fum" };

s1 == s2

s1 == c3

c3 == s1
```


More examples of operator overloading for class String

```
class String
{
public:
   …
   String& operator=(const char*) &;                          // type converting assignment
   String& operator=(std::initializer_list<char>) &;

   char& operator[](size_type);
   char  operator[](size_type) const;

   String& operator+=(const String&);
   …
};

String  operator+(const String&, const String&);

ostream& operator<<(ostream&, const String&);
```


Overloading `operator[]`

To be able to operate on both variable and constant objects, the indexing operator must be overloaded in two versions, non-**const** and **const**.

```
class String
{
public:
   …
   char& operator[](size_type);            // for String

   char  operator[](size_type) const;      // for constant String
   …
};

String       s{ "foobar" };
const String cs{ s };

s[i] = s[i + 1];                        // non-const-version used in both places

s[i] = cs[i];                           // const-version used for cs
```

Implementation does not differ, only the return type and **const**.

Overloading operator<<

Printing a String:

```
String s{ "foobar" };

cout << s << endl;
```

- **operator**<< can not be a member if we want to use infix notation, which we of course do.
  - left operand is an ostream, so it cannot be a member.
- built-in **operator**<< for **const char*** is used to implement.
- public member function c_str() is available, so **friend** can be avoided.

```
ostream& operator<<(ostream& os, const String& str)
{
   return os << str.c_str();
}
```

This signature for **operator**<< can be regarded as an idiom for overloading **operator**<< for an out stream an a user defined type T:

```
ostream& operator<<(ostream& os, const T& t);
```

Guidelines for making an operator function member or non-member

- **If** the operator is one of the foolowing, *it cannot be overloaded*

```
.      .*     :      ::
```

- **If** the operator is one of, it *must be member*

```
=      ->     []     ()
```

- **If** the operator
  1. can have another type as left-hand side argument, or
  2. can have type conversion for its left-hand side argument, or
  3. can be implemented only by using the class' public interface,

  make it a *non-member*, and, if needed in case 1 and 2, also make it *friend*.
- **If** it needs to behave *virtually*, add a virtual member function and implement it in terms of that member function (of interest for polymorphic classes)
- **Otherwise**, let the operator be a *member*.
  - but the following operators are natural to declare as members, since an object of the type in question should be left argument

```
*=     /=     %=     +=     -=     &=     |=     ^=     <<=    >>=    ++     --
```


Some recommendations concerning operator overloading

- preserve natural semantics for operator functions
  - follow the same semantics as their built-in equivalents, whenever not contradicted
- take parameters appropriately by *value*, *reference*, or *const reference*
- choose return type with extra care, if returning a class type
  - *lvalue* or *rvalue* semantics?
  - return *object* or *reference*? **const** or **non**-*const*?
- avoid overloading `&&`, `||` and `,` (comma operator)
  - built-in versions of these enjoys special treatment by the compiler
  - user defined overloadings will be ordinary functions with very different semantics
- arithmetic and assignment operators comes in pair
  - if you overload `+`, you should also overload `+=`
  - they should be defined so that `a+=b` and `a=a+b` have the same meaning
  - a way to achieve this is to define `+` in terms of `+=`
- increment and decrement operators (++ and --)
  - both prefix form and postfix form (latter have an **`int`** dummy parameter) should be defined
  - define the postfix form in terms of the prefix form
  - prefer using the prefix form, if possible
- consider overloading to avoid implicit type conversions
- don't write code that depends on the evaluation order of arguments