C++ Programming I

Operators Types & Operator Overloading

C++ Programming
April 26, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- Operator Overloading

Agenda

- **Operator Overloading**
- **Unary Operators**
  - Unary Increment/Decrement Operator
  - Conversion Operator
  - Dereference & Selection Operator

Agenda

- **Operator Overloading**
- **Unary Operators**
  - Unary Increment/Decrement Operator
  - Conversion Operator
  - Dereference & Selection Operator
- **Binary Operators**
  - Binary Addition `a+b`
  - Addition Assignment `+=`
  - Equality `==`
  - Equality `==`
  - Copy Assignment
  - Subscript Operator
  - Move Assignment
  - Not Overloadable

Operator Overloading

Operators in C++

Intro

- In addition to encapsulating data and methods, classes can also encapsulate **operators**
- You can use these operators to perform operations on your own defined data types, `i.e.` classes
- Operators can be overloaded similarly as functions
- An **operator declaration** looks similar to a function declaration:

```
1 // Operator declaration
2 return_type operator operator_symbol (...parameter list...);
```

- The `operator_symbol` can be any operator types such as `+, -, *, &&` etc.
- Operators provide a more convenient way to work with user defined higher data types as functions do!

Operators in C++

Intro

- Consider the user defined `date` class:

```
1 // User defined data class
2 Date holiday (26, 04, 2018);  // Initialized to April 26, 2018
```

- Which option is more convenient to add a day?

Operators in C++

Intro

- Consider the user defined `date` class:

```
1 // User defined data class
2 Date holiday (26, 04, 2018);  // Initialized to April 26, 2018
```

- Which option is more convenient to add a day?
  1. Option 1 (using the increment operator):
     `++ holiday;`
  2. Option 2 (using a member function Increment()) :
     `holiday.increment();`

Operators in C++

Intro

- Consider the user defined `date` class:

```
1 // User defined data class
2 Date holiday (26, 04, 2018);  // Initialized to April 26, 2018
```

- Which option is more convenient to add a day?
  1. Option 1 (using the increment operator):
     `++ holiday;`
  2. Option 2 (using a member function Increment()) :
     `holiday.increment();`
- For me it's Option 1

Operators in C++

Intro

- Consider the user defined `date` class:

```
1 // User defined data class
2 Date holiday (26, 04, 2018);  // Initialized to April 26, 2018
```

- Which option is more convenient to add a day?
  1. Option 1 (using the increment operator):
     `++ holiday;`
  2. Option 2 (using a member function Increment()) :
     `holiday.increment();`
- For me it's Option 1
- Implementing operator (`<`) in class `Date` for example would help you to compare two dates in an intuitive way:

```
1 if(date1 < date2)
2 {
3     // Do something
4 }
5 else
6 {
7     // Do something else
8 }
```

Unary Operators

Unary Operators

Declaration of Unary Operators

- As the name suggests, operators that function on a single operand are called `unary operators`
- As a static function or implemented in the global namespace, the structure is given by:

```
1 return_type operator operator_type (parameter_type)
2 {
3     // ... implementation
4 }
```

- As a class member, the structure is missing in parameters, because the single parameter that it works upon is the instance of the class itself (`*this`):

```
1 return_type operator operator_type ()
2 {
3 // ... implementation
4 }
```

Unary Operators

List of Unary Operators

- The unary operators that can be overloaded (or redefined) are shown in the table:

Operator	Name
++	Increment
−−	Decrement
*	Pointer dereference
→	Member selection
!	Logical NOT
&	Address-of
~	One's complement
+	Unary plus
−	Unary negation
`Conversion Operators`	

Unary Operators

Prefix and Postfix Operator

- To illustrate the mechanism of operator overloading we implement the prefix increment operator (++)

```
1  // Unary increment operator (prefix)
2  Date& operator++()
3  {
4      // operator implementation code
5      return *this;
6  }
```

Unary Operators

Prefix and Postfix Operator

- To illustrate the mechanism of operator overloading we implement the prefix increment operator (++)

```
1  // Unary increment operator (prefix)
2  Date& operator++()
3  {
4      // operator implementation code
5      return *this;
6  }
```

- The postfix increment operator (++), on the other hand, has a different return type and an input parameter:

```
1  Date operator++(int unused)
2  {
3      // Store a copy before incrementing day
4      Date copy(*this);
5
6      // increment implementation code
7      // Return state before increment (because, postfix)
8      return copy;
9  }
```

Unary Operators

Prefix and Postfix Operator in Class `Data`

```
1  class Date
2  {
3  private:
4      int m_day, m_month, m_year;
5
6  public:
7      Date (int day, int month, int year)
8          : m_day (day), m_month(month), m_year (year) {};
9
10     Date& operator++() // prefix increment
11     {
12         ++m_day;
13         return *this;
14     }
15
16     Date operator++(int) // postfix increment
17     {
18         Date copy(m_day, m_month, m_year);
19         ++m_day;
20         return copy;
21     }
22
23     void displayDate()
24     {
25         std::cout << m_day<< " / " << m_month << " / " << m_year
               << std::endl;
26     }
27 };
```

Unary Operators

Prefix and Postfix Operator in Class `Data`

```
1  int main ()
2  {
3      Date holiday (26, 04, 2018); // April 26, 2018
4
5      std::cout << "The date object is initialized to: ";
6      holiday.displayDate();
7
8      ++holiday; // move date ahead by a day
9      std::cout << "Date after prefix-increment is: ";
10     holiday.displayDate();
11
12     return 0;
13 }
14 // Output
15
16 The date object is initialized to: 26 / 04 / 2018
17 Date after prefix-increment is: 27 / 04 / 2018
```

- The prefix and postfix decrement operators have a similar syntax as the increment operators, just that the declaration would contain a `--` where you see a `++`
- Note: The implementation of class `Date` is reduced to a minimum, *i.e.* month wrapping etc. is not implemented

Conversion Operator Operators

Programming Conversion Operators

- Sometimes you might want to use something like:
  `cout « holiday`
- The code would result in the following compile failure: *error: binary «: no operator found which takes a right-hand operand of type 'Date' (or there is no acceptable conversion)*.

Conversion Operator Operators

Programming Conversion Operators

- Sometimes you might want to use something like:
  `cout « holiday`
- The code would result in the following compile failure: *error: binary «: no operator found which takes a right-hand operand of type 'Date' (or there is no acceptable conversion)*.
- We know that `cout` can work well with a `const char*` :
  `std::cout « "Hello world";` // const char* works!
- Therefore we can add a conversion operator:

```
1 operator const char*()
2 {
3     // operator implementation that returns a char*
4 }
```

Conversion Operator Operators

Programming Conversion Operators

- We can add an operator returning a `const char*`

```
1  #include <sstream> // new include for ostringstream
2
3  operator const char*()
4  {
5      std::ostringstream formattedDate; // for string construction
6      formattedDate << m_day <<" / "<< m_month <<" / "<< m_year;
7      m_outDate = formattedDate.str(); // create copy to member!
8      return m_outDate.c_str(); // return const char*
9  }
```

- Now we can use the following:
  `cout « "Holiday is on:  " « Holiday « endl;`
- The compiler **automatically** uses the output of the appropriate operator in feeding it to `cout` that displays the date on the screen
- Create a copy of `formattedDate` to a member, since the local variable is destroyed when the operator returns!

Dereference & Selection Operator

Smart Pointers

- The dereference operator (*) and member selection operator ($\rightarrow$) are most frequently used in the programming of *smart pointer* classes
- *Smart pointers* are utility classes that wrap regular pointers and simplify memory management by resolving ownership and copy issues using operators
- You will implement your own template based *smart pointer* in one of the following exercises!

Binary Operators

Binary Operators

Declaration of Binary Operators

- Operators that function on two operands are called `binary operators`
- The definition of a binary operator implemented as a global function or a static member function is the following:

```
1 return_type operator operator_type (parameter1, parameter2)
2 {
3     // ... implementation
4 }
```

- Since one parameter is given by class instance itself (`*this`) definition of a binary operator implemented as a class member is:

```
1 return_type operator operator_type (parameter)
2 {
3     // ... implementation
4 }
```

Binary Operators

ist of Binary Operators

- Some binary operators that can be overloaded (or redefined) are shown in the table:

Operator	Name
$+$	Addition
$+=$	Addition/assignment
$-$	Subtraction
$*$	Multiplication
$\div$	Division
$==$	Equality
$=$	Assignment, Copy Assignment and Move Assignment
$->*$	Pointer-to-member selection
$\|\|$	Logical OR
[]	Subscript operator
...	...

- Note: There are many more!

Binary Operators

Binary Addition `a+b` for `Date`

- Imaging you want to add days like this:

```
1 Date Holiday 21, 07, 2018);
2 Date NextHoliday(Holiday + 6);
```

Binary Operators

Binary Addition `a+b` for `Date`

- Imaging you want to add days like this:

```
1 Date Holiday 21, 07, 2018);
2 Date NextHoliday(Holiday + 6);
```

- We can overload the binary addition (+) for `Date`

```
1 Date operator+(int daysToAdd) // binary addition
2 {
3     Date newDate(m_day + daysToAdd, m_month, m_year);
4     return newDate;
5 }
```

- Note: The (−) operator is overloaded similarly

Lecture 6

Dr. P. Arnold

Bern University of Applied Sciences

Binary Operators

Binary Addition `a+b` for `MyString`

- Imaging you want to concatenate strings like this:

```
1 MyString Hello("Hello ");
2 MyString World(" World");
3 MyString HelloWorld(Hello + World); // error: operator+ not
      defined
```


Binary Operators

Binary Addition `a+b` for `MyString`

- Imaging you want to concatenate strings like this:

```
1  MyString Hello("Hello ");
2  MyString World(" World");
3  MyString HelloWorld(Hello + World); // error: operator+ not
       defined
```

- We can overload the binary addition (+) for `MyString`

```
1  MyString operator+(const MyString& addThis)
2  {
3      MyString newString;
4      if (addThis.buffer != nullptr)
5      {
6          newString.buffer = new char[GetLength() +
               strlen(addThis.buffer) + 1];
7          strcpy(newString.buffer, buffer);
8          strcat(newString.buffer, addThis.buffer);
9      }
10     return newString;
11 }
```

- Note: `GetLength()` is a member function of `MyString`

Binary Operators

Addition Assignment +=

- If you provide a + operator the user expects += to work as well!
- Here's the code:

```
1  void operator+=(int daysToAdd) // addition assignment
2  {
3      m_day += daysToAdd;
4  }
```

- Similar implementation for Subtraction Assignment (−=)

Binary Operators

Equality == & Inequality !=

- In the absence of an equality operator == , the compiler simply performs a binary comparison

```
1  if (date1 == date2)
2  {
3      // Do something
4  }
5  else
6  {
7      // Do something else
8  }
```


Bern University
of Applied Sciences

Binary Operators

Equality == & Inequality !=

- In the absence of an equality operator == , the compiler simply performs a binary comparison

```
1  if (date1 == date2)
2  {
3      // Do something
4  }
5  else
6  {
7      // Do something else
8  }
```

- The general form to overload the equality (==) and inequality (!=) operators is:

```
1  bool operator== (const ClassType& compareTo)
2  {
3      // comparison code here, return true if equal else false
4  }
5
6  bool operator!= (const ClassType& compareTo)
7  {
8      // comparison code here, return true if inequal else false
9  }
```

- Note: The inequality operator can reuse the equality operator

Binary Operators

Equality == & Inequality !=

- For class `Date` this yields:

```
1  bool operator==(const Date& compareTo)
2  {
3      return ((m_day == compareTo.m_day)
4              && (m_month == compareTo.m_month)
5              && (m_year == compareTo.m_year));
6  }
7
8  bool operator!=(const Date& compareTo)
9  {
10     return !(this->operator==(compareTo));
11 }
```

- The inequality operator can reuse the equality operator
- Note: Similar implementation for operators `< , <= , >, >=`

Binary Operators

Copy Assignment =

- Sometimes you want to assign the contents of an instance of a class to another:

```
1 Date holiday(15, 07, 2018);
2 Date anotherHoliday(12, 08, 2018);
3 anotherHoliday = holiday; // uses copy assignment operator
```

- This assignment invokes the default copy assignment operator that the compiler has built in to your class when you have not supplied one
- This problem with the default copy assignment operator is similar to the one with the default copy constructor discussed earlier!
- To ensure deep copies you have to supply your own version

```
1 ClassType& operator=(const ClassType& copySource)
2 {
3     if(this != &copySource) // protection against copy into self
4     {
5         // copy assignment operator implementation
6     }
7     return *this;
8 }
```

Note:

Deep copies are important if your class encapsulates a raw pointer!

Binary Operators

Copy Assignment = for `MyString`

- For `MyString` the implementation will look like this:

```
1  MyString& operator=(const MyString& copySource)
2  {
3      if ((this != &copySource) && (copySource.buffer != nullptr))
4      {
5          if (buffer != nullptr)
6              delete[] buffer;
7
8          // ensure deep copy by first allocating own buffer
9          buffer = new char [strlen(copySource.buffer) + 1];
10
11         // copy from the source into local buffer
12         strcpy(buffer, copySource.buffer);
13     }
14     return *this;
15 }
```

1. Check that copy source is not destination
2. Deallocates its internal buffer
3. Reallocating space

Note:

When your class encapsulates a raw pointer write **CTor, DTor, CopyCTor and copy assignment operator!**

Binary Operators

Subscript Operator []

- The typical syntax of a subscript operator is:

```
1 return_type& operator[](subscript_type& subscript);
```


Binary Operators

Subscript Operator []

- The typical syntax of a subscript operator is:

```
return_type& operator[](subscript_type& subscript);
```

- For `MyString` that yields:

```
const char& operator[](int index) const
{
    if (index < GetLength())
        return buffer[index];
}
```

Binary Operators

Subscript Operator []

- The typical syntax of a subscript operator is:

```
return_type& operator[](subscript_type& subscript);
```

- For `MyString` that yields:

```
const char& operator[](int index) const
{
    if (index < GetLength())
        return buffer[index];
}
```

- Using keyword `const` is important even when programming operators

```
MyString sayHello("Hello World");
sayHello[2] = 'k'; //error: operator[] is const
```

- By using `const char&` you are protecting internal member `MyString::buffer` from direct modifications from the outside via `operator[]`
- In addition the operator function type is `const` to ensure that it cannot modify the class member attributes.

Binary Operators

Move Assignment Operator and Move Constructor

- The move constructor and the move assignment operators are performance optimization features since C++ 11
- They ensuring that temporary values (rvalues that don't exist beyond the statement) are not wastefully copied!
- Check the following code for an example

```
1 MyString Hello("Hello ");
2 MyString World("World");
3 MyString CPP(" of C++");
4 MyString sayHello(Hello + World + CPP); // operator+, copy cTor
5 MyString sayHelloAgain ("overwrite this");
6 sayHelloAgain = Hello + World + CPP;
7 // operator+, copy constructor, copy assignment operator=
```

- To enable this functionality the the binary addition operator + is used

```
1 MyString operator+(const MyString& addThis)
2 {
3     MyString newStr;
4     if (addThis.buffer != nullptr)
5     {
6         // copy into newStr
7     }
8     return newStr; // return copy by value, invoke copy cTor
9 }
```

Lecture 6

Dr. P. Arnold

Bern University of Applied Sciences

Binary Operators

Move Assignment Operator and Move Constructor

- To enable more efficient code a move constructor and a move assignment operator have to be declared:

```
1  MyString(MyString&& moveSrc) // move constructor
2  {
3      cout << "Move constructor moves: " << moveSrc.m_buffer <<
           endl;
4      if(moveSrc.m_buffer != nullptr)
5      {
6          m_buffer = moveSrc.m_buffer; // take ownership i.e.
               'move'
7          moveSrc.m_buffer = nullptr;    // free move source
8      }
9  }
10
11 MyString& operator=(MyString&& moveSrc) // move assignment op.
12 {
13     cout << "Move assignment op. moves: " << moveSrc.buffer <<
           endl;
14     if((moveSrc.m_buffer != nullptr) && (this != &moveSrc))
15     {
16         delete[] m_buffer; // release own buffer
17         m_buffer = moveSrc.m_buffer; // take ownership i.e.
18         moveSrc.m_buffer = nullptr;    // free move source 'move'
19     }
20     return *this;
21 }
```

- See book for complete implementation!

Bern University of Applied Sciences

Binary Operators

Move Assignment Operator and Move Constructor

- Comparing the output and performance:

```
1  // Output without move
2  Constructor called for: Hello
3  Constructor called for: World
4  Constructor called for: of C++
5  Constructor called for: overwrite this
6  operator+ called:
7  Default constructor called
8  Copy constructor copies: Hello World // -> 1 Copy
9  operator+ called:
10 Default constructor called
11 Copy constructor copies: Hello World of C++ // -> 2 Copy
12 Copy assignment op. copies: Hello World of C++ // -> 3 Copy
13
14 // Output with move
15 Constructor called for: Hello
16 Constructor called for: World
17 Constructor called for: of C++
18 Constructor called for: overwrite this
19 operator+ called:
20 Default constructor called
21 Move constructor moves: Hello World // -> Move
22 operator+ called:
23 Default constructor called
24 Move constructor moves: Hello World of C++ // -> Move
25 Move assignment op. moves: Hello World of C++ // -> Move
```

- Supplying move cTor and the move assignment operator is optional

Operators

Operators That CANNOT Be Overloaded or Redefined

- The operators that cannot be redefined are shown in the table:

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
?	Conditional ternary operator
sizeof	Gets the size of an object/class type

Thank You

Questions

???