CSE 1325: Object-Oriented Programming

Lecture 21

# C++ Operator Overloading


## Mr. George F. Rice

george.rice@uta.edu


**Office Hours:**
**Prof Rice 12:30 Tuesday and Thursday in ERB 336**
**For TAs see this web page**

My English teacher demanded that I name two pronouns.
I exclaimed, "Who, me?"

# Today's Topics

- Operator Overloading Theory and Examples
  - Stream out (<<) and in (>>)
  - Comparisons (<=> spaceship or ==, !=, <, <=, >, >=)
  - Math (+, +=, and ++)
- Optional Topics
  - Regular expressions (regex) for use with the streaming in operator >>
  - Polymorphism with functions: A strategy

# Should Our Own Classes Be 2nd Class Classes?

```
std::cout << _day << ' '
          << to_string(_month) << ", "
          << _year << std::endl;
birthday.print_date();
```

The other types just stream to std::cout directly.

But our Month and Date types must be converted into a string, via a helper function or method! *Unfair*!

We want our Month and Date classes to be a 1st class types! Like this:

```
std::cout << month << std::endl;
std::cout << birthday << std::endl;
```

**We need to overload the << operator for our Month and Date types!**

# Operator Overloading

- We can define the “<<” operator for our Month and Date classes via Operator Overloading

**Operator Overloading** - Providing a user-defined meaning to a pre-defined operator (such as <<, >>, ==, and +) for a user-defined type (class or enum).


- Most C++ operators can be overloaded
  - The key is to know the method or function signature, e.g., the parameter types and return values
    - https://en.wikipedia.org/wiki/Operators_in_C_and_C++
  - Let’s teach Date (and Month) some operators!

# Theory of Operator Overloading

- An infix operator is just a 1-parameter method or 2-parameter function
  - Consider `a=b+c; // a, b, c, d are ints`
  - This is the same as `a=b.operator+(c);`
    `// the method name is operator+`
  - Also the same as `a=operator+(b, c);`
    `// the function name is operator+`
  - `a=b+c+d;` is the same as `a = b.operator+(c.operator+(d));`
  - `std::cout << i;` is the same as `operator<<(std::cout, i);`
    `std::cout << i << j;` is the same as
    `operator<<(operator<<(std::cout, i), j);`
- A unary operator is just a parameterless method or 1-parameter function
  - `++a` is just `a.operator++()` or `operator++(a)`

# Enum Class Month Output Stream if

- We define the << *function* for our Month enum in terms of << that have already been defined
- Just stream out the std::string version as if writing to std::out

```
enum class Month {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};
```

**month.h**

const reference – NOT changed!

```
std::ostream& operator<<(std::ostream& ost, const Month& month) {
    if(month == Month::Jan) ost << "January";
    if(month == Month::Feb) ost << "February";
    if(month == Month::Mar) ost << "March";
    if(month == Month::Apr) ost << "April";
    if(month == Month::May) ost << "May";
    if(month == Month::Jun) ost << "June";
    if(month == Month::Jul) ost << "July";
    if(month == Month::Aug) ost << "August";
    if(month == Month::Sep) ost << "September";
    if(month == Month::Oct) ost << "October";
    if(month == Month::Nov) ost << "November";
    if(month == Month::Dec) ost << "December";
    return ost;
}
```

**month.cpp**

A std::ostream is an “output stream”. Think std::cout, our most famous ostream!

C++ already knows how to stream a std::string, so we’re set!

Don’t forget to return the std::ostream unless you like segfaults!

# Enum Class Month Output Stream
# switch

- Switch is a natural choice for enum classes

month.h

```
enum class Month {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};
```

month.cpp

```
std::ostream& operator<<(std::ostream& ost, const Month& month) {
    switch(month) {
        case Month::Feb: ost << "February"; break;
        case Month::Mar: ost << "March"; break;
        case Month::Apr: ost << "April"; break;
        case Month::May: ost << "May"; break;
        case Month::Jun: ost << "June"; break;
        case Month::Jul: ost << "July"; break;
        case Month::Aug: ost << "August"; break;
        case Month::Sep: ost << "September"; break;
        case Month::Oct: ost << "October"; break;
        case Month::Nov: ost << "November"; break;
        case Month::Dec: ost << "December"; break;
    }
    return ost;
}
```

# Enum Class Month Output Stream
# map

- Maps are often useful in simplifying streams
  - The map does the switch implicitly

month.h

```
enum class Month {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};
```

month.cpp

```
const static std::map<Month, std::string> month_to_string {
    {Month::Jan, "January"},  {Month::Feb, "February"},  {Month::Mar, "March"},
    {Month::Apr, "April"},    {Month::May, "May"},       {Month::Jun, "June"},
    {Month::Jul, "July"},     {Month::Aug, "August"},    {Month::Sep, "September"},
    {Month::Oct, "October"},  {Month::Nov, "November"},  {Month::Dec, "December"},
};

std::ostream& operator<<(std::ostream& ost, const Month& month) {
    return ost << month_to_string.at(month); // throws std::out_of_range if invalid
}
```

Note that although month_to_string is in the global namespace, it is NOT accessible outside date.cpp – because you CANNOT include date.cpp, right?

This is sometimes called the “file pseudo-scope”. Names defined in a .cpp file are ONLY visible within that file, NEVER elsewhere. Convenient!

# Class Date Output Stream

- For classes as with enums, we ALWAYS declare our operator<< as a *function*
  - We'll discuss why later
- Operator<< is USUALLY a friend, so it can access the fields directly
  - Although if getters or a format method are available, it can be an enemy instead :)
- Declare the friend function like this

**date.h**

```
class Date {
  public:
    Date(int year, Month month, int day)
       : _year{year}, _month{month}, _day{day} {
        if (1 > day || day > 31) throw std::runtime_error{"Invalid day"};
    }
    friend std::ostream& operator<<(std::ostream& os, const Date& date);
  private:
    int _year;
    Month _month;
    int _day;
};
```

Operator<< is Date's *friend*. That simply means this function can access Date's private data: _day, _month, and _year.

# Date (Class) Output Stream

- The implementation is then straightforward
- Just decide how you'd format the output to **`std::cout`**, but write to the **`std::ostream& ost`** instead
  - Remember, this is a function NOT a method. You need **`date.`** before the fields!

**date.cpp**

```
std::ostream& operator<<(std::ostream& ost, const Date& date) {
  ost << date._year << " " << date._month << " " << date._day;
  return os;
}
```

And don't forget to return that std::ostream!

- Our class is a first-class class at last!

```
int main() {
    Date birthday{1950, Month::Dec, 30};
    std::cout << birthday << std::endl;
}
```

This really cleans up our main code!

```
ricegf@antares:~/dev/202301/20/code_from_slides/operator_overloading$ ./birthday
1950 December 30
```

# Overloading Stream In

- We also want to be able to read in our Month and Date objects as easily as int and double
  - We need to overload **`operator>>`**
- Streaming in the Month enum class is a challenge
  - We'll need to stream in a std::string, then parse it
  - But what's "valid" input?
    - We could require a numeric (12) string or name string ("Dec")
    - Or we could use a *regular expression* (regex) to recognize *many* forms of month: "December", "Dec", "12", "December,", and similar combinations
      - See the Appendix for C++ regular expressions – NOT on the exam, guaranteed!
- We also need data validation, including **`days_in_month`**
  - Throw exception on 2019 February 31, 1900 February 29, or other invalid dates

# Stream In a Month

- Maps easily convert **`std::string`** to **`Month`**, too
  - Not quite as flexible as a regular expression
  - But will convert “Dec” to Month::December – and other code handles “12”

**date.cpp**

```
const static std::map<std::string, Month> month_records = {
    {"jan", Month::Jan}, {"feb", Month::Feb}, {"mar", Month::Mar},
    {"apr", Month::Apr}, {"may", Month::May}, {"jun", Month::Jun},
    {"jul", Month::Jul}, {"aug", Month::Aug}, {"sep", Month::Sep},
    {"oct", Month::Oct}, {"nov", Month::Nov}, {"dec", Month::Dec},
};

std::istream& operator>>(std::istream& is, Month& month) {
  std::string s; is >> s;                       // Read a string
  if(isdigit(s[0])) {                           // Numeric?
      int index = stoi(s) - 1;                      //Convert to index
      if(index < 0 || index > 11)                   // Verify range
          throw std::out_of_range{"Invalid month: " + s};
      month = (Month) index;                        // Set month
  } else {                                      // String?
      for(char& c : s) c = (char) tolower(c);       // Force all lower case
      month = month_records.at(s);                  // Set month – if not found,
                                                    //   will throw std::out_of_range
  }
  return is;
}
```

reference – *will* be changed!

Don’t forget to return the std::istream!

# Validating Days in a Month

- To ensure we have a valid **`_day`**, we need to know how many days are in each month
  - Which, thanks to the ancient Romans and February, is *complicated*

```
int Date::days_in_month(Month month, int year) {
    switch(month) {
      case Month::Apr: // old-school “or”
      case Month::Jun:
      case Month::Sep:
      case Month::Nov: return 30;
      case Month::Feb: return ((year%400==0) || (year%4==0 && year%100!=0)) ? 29 : 28;
      default:         return 31;
  }
}
```

**date.cpp**

When orbital mechanics intrude on “elegant code”…

# Stream In a Date

- Given Month's `>>` and the **`days_in_month`** method, we can stream in a date!

**date.h**

```
class Date {
  private:
    int days_in_month(Month month, int year); // See previous slide
```

**date.cpp**

```
Date::Date(int year, Month month, int day) : _year{year}, _month{month}, _day{day} {
    if (day < 1 || day > days_in_month(month, year))
        throw std::runtime_error{"Invalid day: " + std::to_string(day)};
}
Date::Date() : Date{1970, Month::Jan, 1} { }

std::istream& operator>>(std::istream& is, Date& date) {
  is >> date._year;
  is >> date._month;
  is >> date._day;
  if (1 > date._day || date._day > date.days_in_month(date._month, date._year))
      throw std::runtime_error{"Invalid day"};
  return is;
}
```

Add a default constructor so we can simply "`date d;`"

Notice we read the Month *exactly* as we read each int!

This is a *friend function*, NOT a method, so you must specify the object, too!

# Date's Stream In Like Other Types

- Now reading a date is as easy as an int!

pick_a_date.cpp

```
#include "date.h"

int main() {
    Date date;
    while(std::cin) {
        std::cout << "\nEnter a valid date! (year month day): ";
        try {
            std::cin >> date;
            std::cout << "You entered " << date << std::endl;
        } catch(...) {
            std::cerr << "That wasn't a valid date!" << std::endl;
        }
    }
}
```

# Date's Stream In Like Other Types!

```
ricegf@antares:~/dev/202408/21-c++-op-overloading/code_from_slides$ ./pick_a_date

Enter a valid date! (year month day): 2023 Apr 5
You entered 2023 April 5

Enter a valid date! (year month day): 2023 4 5
You entered 2023 April 5

Enter a valid date! (year month day): 2023 04 5
You entered 2023 April 5

Enter a valid date! (year month day): 2023 Apr 31
That wasn't a valid date!

Enter a valid date! (year month day): 2019 Feb 29
That wasn't a valid date!

Enter a valid date! (year month day): 2020 Feb 29
You entered 2020 February 29

Enter a valid date! (year month day): 2020 February 29
That wasn't a valid date!

Enter a valid date! (year month day):
```

# Why must << and >> be functions?

- Consider ==
  - **Function**: `bool operator==(Date& date1, Date& date2);`
  - **Method**: `bool Date::operator==(Date& date2);`
  - For the method, `date1` (the first parameter) is the current object
- Consider <<
  - Function: `ostream& operator<<(ostream& os, Date& date);`
  - Method: `ostream& operator<<(Date& date);`
  - For the method, `os` (the first parameter – the **STREAM**) is the current object
  - **We can't redefine `ostream`** – the first parameter – to handle date! The method version of operator<< and >> are on the "wrong" object.
  - Thus, **we can ONLY use the function form for << and >>**, never the method form

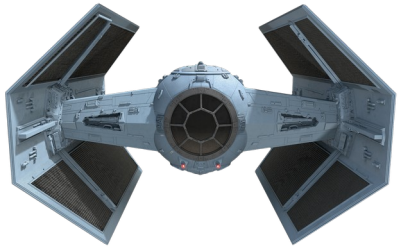
# Comparing Objects

- In Java, the == compares object *addresses*
  - The **`.equals`** method by *default* does, too!
  - But we *override* it to compare values
- In C++, the == is *not defined* for objects
  - But we can overload it (and !=, <, <=, >, and >=) to compare values!

# The “Spaceship” (<=>) Operator

## in C++ 20 and Later

- The obvious default would be to compare each field in order of declaration – simple!
- C++ 20 can do this... *if you ask it nicely*
  - We just declare the “spaceship operator” (<=>)
  - Earlier versions of C++ required additional code – next slide!



date.h

```
class Date {
  public:
    Date(int year = 1970, Month month = Month::Jan, int day = 1);

    auto operator<=>(const Date&) const = default;
    // NOTHING is required in the .cpp file!
```

Star Wars Tie Fighter model by Jansen_G via the Pixabay License
https://www.freeimg.net/photo/1141024/spaceship-model-toys-starwars

# The "Spaceship" (<=>) *Approach*

## in C++ 17 and Earlier AND When Not Every Field is Relevant

**In C++ 17 and earlier, we write our own "spaceship"!**

Define all operators using the private compare() method

**date.h**

```
inline bool operator==(const Date& rhs) const {return (compare(rhs) == 0);}
inline bool operator!=(const Date& rhs) const {return (compare(rhs) != 0);}
inline bool operator< (const Date& rhs) const {return (compare(rhs) <  0);}
inline bool operator<=(const Date& rhs) const {return (compare(rhs) <= 0);}
inline bool operator> (const Date& rhs) const {return (compare(rhs) >  0);}
inline bool operator>=(const Date& rhs) const {return (compare(rhs) >= 0);}
```

The operators match!

Date::compare returns -1 if `this < rhs`, 0 if `this == rhs`, and 1 if `this > rhs`

```
int Date::compare(const Date& rhs) const {
    if(year <rhs.year ) return -1;
    if(year >rhs.year ) return  1;
    if(month<rhs.month) return -1;
    if(month>rhs.month) return  1;
    if(day  <rhs.day  ) return -1;
    if(day  >rhs.day  ) return  1;
    return 0;
}
```

The compare method returns -1 if object is less, 0 if equal, or 1 if greater than its parameter. (Yes, it's Java's compareTo in disguise!)

**date.cpp**

**Inline** tells the compiler to replace any call to these methods with the literal code instead of a function call and return.

# The “Spaceship” (<=>) *Approach*

## if you have no idea if it’s supported!

- We can use the preprocessor to check at compile time whether <=> is supported
  - Use the spaceship if it’s available, otherwise define all operators using the private compare() method
  - You may also wrap the compare method declaration and definition in the same preprocessor statement to (slightly) reduce code size

**date.h**

```
#ifdef __cpp_impl_three_way_comparison
    auto operator<=>(const Date&) const = default;
#else
    inline bool operator==(const Date& rhs) {return (compare(rhs) == 0);}
    inline bool operator!=(const Date& rhs) {return (compare(rhs) != 0);}
    inline bool operator< (const Date& rhs) {return (compare(rhs) <  0);}
    inline bool operator<=(const Date& rhs) {return (compare(rhs) <= 0);}
    inline bool operator> (const Date& rhs) {return (compare(rhs) >  0);}
    inline bool operator>=(const Date& rhs) {return (compare(rhs) >= 0);}
#endif
```

# The != Operator
## in C++ 20 and Later

- Prior to C++ 20, you had to override all 6 comparison operators to support all comparisons: ==, !=, <, <=, >, >=
- In C++ 20, if you DON'T use the spaceship <=>, you need only override 5 comparison operators: ==, <, <=, >, >=
  - A != B now defaults to !(A == B) if not defined
- I have no idea why they stopped† there – given definitions for == and <
  - A != B is just !(A == B)
  - A >= B is just !(A < B)
  - A > B is just !(A == B) && !(A < B)
  - A <= B is just (A == B) || (A < B)
- But who asked me?

# Date and Month Regression Test

test_date.cpp

```
int main() {
    int result = 0;
    int vector = 1;

    // Test ==, !=, <, <=, >, >=
    try {
        Date lesser {2023, Month::Apr, 5};
        Date greater{2023, Month::Apr, 6};

        if(lesser == greater) {
            result |= vector;
            std::cerr << "FAIL: Invalid == operator for"
                      << "\n  " << lesser << " == " << greater
                      << std::endl;
        }
        if(!(lesser != greater)) {
            result |= vector;
            std::cerr << "FAIL: Invalid != operator for"
                      << "\n  ! " << lesser << " != " << greater
                      << std::endl;
        }

        // See full regression test at cse1325-prof/21-C++-op-overloading!
```

```
ricegf@antares:~/dev/202301/20/code_from_slides/operator_overloading$ make test_date
g++ --std=c++17 -c date.cpp -o date.o
g++ --std=c++17 test_date.cpp date.o -o test_date
ricegf@antares:~/dev/202301/20/code_from_slides/operator_overloading$ ./test_date
ricegf@antares:~/dev/202301/20/code_from_slides/operator_overloading$
```

# Other Operators - ++

- In a 3-term for loop iterating over dates, we need to ++ dates
  - ++day and ++year are predefined by int
  - What about ++month? And (for completeness) month++?
    - ++month should advance Month month to the next one in the calendar
    - Roll over from December to January
    - month++ is identical, except we should return the original month rather than the updated month

# Pre- and Post-Incrementing Month

month.h

```
enum class Month {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};
std::ostream& operator<<(std::ostream& ost, const Month& month);
Month& operator++(Month& m); // Pre-increment
Month  operator++(Month& m, int); // Post-increment (the parameter is ignored)
```

month.cpp

```
Month& operator++(Month& m) {   // Pre-increment, e.g., ++m
  switch(m) {
    case Month::Jan: m = Month::Feb; break;
    case Month::Feb: m = Month::Mar; break;
    case Month::Mar: m = Month::Apr; break;
    case Month::Apr: m = Month::May; break;
    case Month::May: m = Month::Jun; break;
    case Month::Jun: m = Month::Jul; break;
    case Month::Jul: m = Month::Aug; break;
    case Month::Aug: m = Month::Sep; break;
    case Month::Sep: m = Month::Oct; break;
    case Month::Oct: m = Month::Nov; break;
    case Month::Nov: m = Month::Dec; break;
    case Month::Dec: m = Month::Jan; break;
  }
  return m;
}
Month operator++(Month& m, int) {  // Post-increment, e.g., m++
  Month result{m};
  ++m;
  return result;
}
```

Must be a function, since enums can't have methods.

Since enum classes aren't ints, incrementing is a bit of a pain.

Or use a map (left as an exercise for the above-average student).

Just like pre-increment, but we have to return the original value!

**Pro Tip:** Do you see why we prefer ++m to m++?

# Pre- and Post-Incrementing Date

(1 of 2)

date.h

```
class Date {
  public:
    Date(int year, Month month, int day);
    friend std::ostream& operator<<(std::ostream& os, const Date& date);
    Date& operator++();    // Pre-increment
    Date  operator++(int); // Post-increment (the parameter is ignored)
  private:
    int _year;
    Month _month;
    int _day;
};
```

Note that these ++ operators are defined as *methods*, since Date is a true class.

# Pre- and Post-Incrementing Date

(2 of 2)

date.cpp

```
Date& Date::operator++() {    // Pre-increment
    ++_day;
  if (_day > days_in_month(_month, _year)) {
    if (_month == Month::Dec) {
      ++_year;
    }
    ++_month;
    _day = 1;
  }
  return *this;
}

Date Date::operator++(int) { // Post-increment (the parameter is ignored)
   Date date{*this};  // a “copy constructor” from this object
   ++*this;
   return date;
}
```

# Pre- and Post-Incrementing Date

# Interactive Test

**interactive_increments.cpp**

```
#include "date.h"
#include <iostream>

int main() {
    Month m{Month::Nov};
    std::cout << m++ << ' ' << m++ << ' '
              << m << ' '
              << ++m << ' ' << ++m << ' ' << std::endl;
    for (Month m=Month::Aug; m != Month::Feb ; ++m) std::cout << m << ' ';
    std::cout << std::endl;
    {
    Date d{1950, Month::Dec, 30}; // Wrap Dec -> Jan
    std::cout << d++ << ' ' << d++ << ' '
              << d << ' '
              << ++d << ' ' << ++d << ' ' << std::endl;
    } {
    Date d{1900, Month::Feb, 27}; // Not leap year (100 year rule)
    std::cout << d++ << ' ' << d++ << ' '
              << d << ' '
              << ++d << ' ' << ++d << ' ' << std::endl;
    } {
    Date d{2000, Month::Feb, 27}; // Leap year (400 year rule)
    std::cout << d++ << ' ' << d++ << ' '
              << d << ' '
              << ++d << ' ' << ++d << ' ' << std::endl;
    } {
    Date d{2001, Month::Feb, 27};  // Not leap year
    std::cout << d++ << ' ' << d++ << ' '
              << d << ' '
              << ++d << ' ' << ++d << ' ' << std::endl;
```

Using Month directly in a for loop!

# Pre- and Post-Incrementing Date

interactive_increments.cpp

```
#include "date.h"
#include <iostream>
```

```
student@cse1325:/media/sf_dev/06$ m op_overload3
i                \\\///
                / _  _ \
              (| (.)(.) |)
  .----------.OOOo--()--oOOO.----------.
  |                                    |
  |                                    |
  |         Starting a Build           |
  |    Fri Jan 25 15:31:41 CST 2019    |
  |                                    |
  |                                    |
  '----------.oooO--------------------'
             (   )   Oooo.
              \ (    (   )
               \_)    ) /
                     (_/
g++ --std=c++17 -c op_overload3.cpp
g++ --std=c++17 -o op_overload3 op_overload3.o date.o
student@cse1325:/media/sf_dev/06$ ./op_overload3
November December January February March
August September October November December January
30 December, 1950 31 December, 1950 1 January, 1951 2 January, 1951 3 January, 1951
27 February, 1900 28 February, 1900 1 March, 1900 2 March, 1900 3 March, 1900
27 February, 2000 28 February, 2000 29 February, 2000 1 March, 2000 2 March, 2000
27 February, 2001 28 February, 2001 1 March, 2001 2 March, 2001 3 March, 2001
27 February, 2004 28 February, 2004 29 February, 2004 1 March, 2004 2 March, 2004
student@cse1325:/media/sf_dev/06$
```

```
    Date d{2001, Month::Feb, 27};
    std::cout << d++ << ' ' << d++ << ' '
              << d << ' '
              << ++d << ' ' << ++d << ' ' << std::endl;
```

# Prefix or Postfix Increment As a Method or Function

- Most overloaded operators in C++ can be method OR function
  - We usually prefer the method approach
  - But... enum classes have no methods :-(
    (did I mention that already?)
- Month and Date *decrement* operators are left as an exercise for the student (it's good practice!)

## Increment/decrement operators

Increment/decrement operators increment or decrement the value of the object.

| Operator name | Syntax | Overloadable | Prototype examples (for `class T`) | |
|---|---|---|---|---|
| | | | Inside class definition | Outside class definition |
| pre-increment | `++a` | Yes | `T& T::operator++();` | `T& operator++(T& a);` |
| pre-decrement | `--a` | Yes | `T& T::operator--();` | `T& operator--(T& a);` |
| post-increment | `a++` | Yes | `T T::operator++(int);` | `T operator++(T& a, int);` |
| post-decrement | `a--` | Yes | `T T::operator--(int);` | `T operator--(T& a, int);` |

http://en.cppreference.com/w/cpp/language/operator_incdec

# Operators Can Accept Any Types
## (except all primitive types)

- Adding two dates makes little sense
  - But **adding a date to an integer n** sounds like "give me a date that is n days later"
  - So Date{1950, Month::Dec, 30} + 5 is Jan 4, 1951

date.h

```
class Date {
  public:
    …
    Date  operator+(int n);   // Number of days past the current date
```

date.cpp

```
Date Date::operator+(int n){  // Number of days past the current date
    Date d{*this};
    for ( ; n>0; --n) ++d;    // Horribly inefficient for large n! But concise…
    return d;
}
```

interactive_increments.cpp

```
    Date d{2001, Month::Jan, 1};
    int n;
    while(true) {
        std::cout << "The date is now " << d << ", add how many days? ";
        std::cin >> n;
        if (n<=0) break;
        d = d + n;
    }
```

# Operators Can Accept Any Types
## (except all primitive types)

- Adding two dates makes little sense
  - But adding a date to an integer n sounds like

```
student@cse1325:/media/sf_dev/06$ make op_overload3
g++ --std=c++17 -c op_overload3.cpp
g++ --std=c++17 -c date.cpp
g++ --std=c++17 -o op_overload3 op_overload3.o date.o
student@cse1325:/media/sf_dev/06$ ./op_overload3
November December January February March
August September October November December January
30 December, 1950 31 December, 1950 1 January, 1951 2 January, 1951 3 January, 1951
27 February, 1900 28 February, 1900 1 March, 1900 2 March, 1900 3 March, 1900
27 February, 2000 28 February, 2000 29 February, 2000 1 March, 2000 2 March, 2000
27 February, 2001 28 February, 2001 1 March, 2001 2 March, 2001 3 March, 2001
27 February, 2004 28 February, 2004 29 February, 2004 1 March, 2004 2 March, 2004
3-term for loop:
27 February, 2004 28 February, 2004 29 February, 2004 1 March, 2004 2 March, 2004
The date is now 1 January, 2001, add how many days? 1
The date is now 2 January, 2001, add how many days? 7
The date is now 9 January, 2001, add how many days? 325
The date is now 30 November, 2001, add how many days? 31
The date is now 31 December, 2001, add how many days? 1
The date is now 1 January, 2002, add how many days? 365
The date is now 1 January, 2003, add how many days? 365
The date is now 1 January, 2004, add how many days? 9
The date is now 10 January, 2004, add how many days? 0
student@cse1325:/media/sf_dev/06$
```

# Are Operators symmetric?

- That is, are “Date + int” and “int + Date” the same?

interactive_increments.cpp

```
while(true) {
    Date d;
    try {
        std::cout << "Enter a starting date (day month, year): ";
        std::cin >> d;
        int n;
        while(true) {
            std::cout << "The date is now " << d << ", add how many days? ";
            std::cin >> n;
            if (n<=0) break;
        //  d = d + n;
            d = n + d;  // Same difference, right?
        }
        break;
    } catch(std::runtime_error e) {
        std::cerr << e.what() << std::endl;  // print runtime_error’s param
        std::cin.ignore(4096, '\n');         // clear cin’s buffer
    }
}
```

# No. No, They Are Not!

- That is, are “Date + int” and “int + Date” the same?

interactive_increments.cpp

```
student@cse1325:/media/sf_dev/06$ make op_overload3
g++ --std=c++17 -g -c op_overload3.cpp
op_overload3.cpp: In function ‘int main()’:
op_overload3.cpp:56:23: error: no match for ‘operator+’ (operand types are ‘int’ and ‘Date’)
                 d = n + d;
                     ~~^~~
Makefile:59: recipe for target 'op_overload3.o' failed
make: *** [op_overload3.o] Error 1
student@cse1325:/media/sf_dev/06@
```

```
    while(true) {
            if (n<=0) break;
        //  d = d + n;
            d = n + d;  // Same difference, right?
        }
        break;
    } catch(std::runtime_error e) {
        std::cerr << e.what() << std::endl;  // print runtime_error’s param
        std::cin.ignore(4096, '\n');         // clear cin’s buffer
    }
}
```

Noooooooo.....

# Operator Symmetry

- Adding the symmetric <u>function</u> (ugh) fixes this

date.h

```
class Date {
  public:
    Date  operator+(int n);   // Number of days past the current date
};
Date operator+(int n, Date& date);  // Symmetry for Date::operator+(int n)
```

date.cpp

```
Date Date::operator+(int n){  // Number of days past the current date
    Date d{*this};
    for ( ; n>0; --n) ++d;    // Horribly inefficient for large n! But concise…
    return d;
}
Date operator+(int n, Date& date) {return date + n;}
```

interactive_increments.cpp

```
    Date d{2001, Month::Jan, 1};
    int n;
    while(true) {
        std::cout << "The date is now " << d << ", add how many days? ";
        std::cin >> n;
        if (n<=0) break;
     // d = d + n;
        d = n + d;
    }
```

# Are Compound Operators Automatic?

interactive_increments.cpp

```
while(true) {
    Date d;
    try {
        std::cout << "Enter a starting date (day month, year): ";
        std::cin >> d;
        int n;
        while(true) {
            std::cout << "The date is now " << d << ", add how many days? ";
            std::cin >> n;
            if (n<=0) break;
        //  d = d + n;
        //  d = n + d;  // Same difference, right?
            d += n;     // Surely OK?
        }
        break;
    } catch(std::runtime_error e) {
        std::cerr << e.what() << std::endl;  // print runtime_error's param
        std::cin.ignore(4096, '\n');         // clear cin's buffer
    }
}
```

# No. No, They Are Not!

interactive_increments.cpp

```
student@cse1325:/media/sf_dev/06$ make op_overload3
g++ --std=c++17 -g -c op_overload3.cpp
op_overload3.cpp: In function 'int main()':
op_overload3.cpp:57:19: error: no match for 'operator+=' (operand types are 'Date' and 'int')
                d += n;
                ~~^~~~
Makefile:59: recipe for target 'op_overload3.o' failed
make: *** [op_overload3.o] Error 1
student@cse1325:/media/sf_dev/06@
```

```
    while(true) {
        Date d;
        try {
            //  d = d + n;
            //  d = n + d;  // Same difference, right?
                d += n;     // Surely OK?
            }
            break;
        } catch(std::runtime_error e) {
            std::cerr << e.what() << std::endl;  // print runtime_error's param
            std::cin.ignore(4096, '\n');         // clear cin's buffer
        }
    }
```

Nope

# Compound Operators

- These must modify the *existing* object

date.h

```
class Date {
  public:
    Date  operator+(int n);   // Number of days past the current date
    Date& operator+=(int n);  // Number of days past the current date
};
```

date.cpp

```
Date Date::operator+(int n){  // Number of days past the current date
    Date d{*this};            // Make a copy of this (copy construtor)
//  for ( ; n>0; --n) ++d;    // Horribly inefficient for large n! But concise…
    d += n;                   // Now rely on += (DRY!)
    return d;
}
Date& Date::operator+=(int n){    // Compound
    for ( ; n>0; --n) ++(*this);  // Horribly inefficient for large n!
    return *this;
}
```

```
27 February, 2000 28 February, 2000 29 February, 2000 1 March, 2000 2 March, 2000
27 February, 2001 28 February, 2001 1 March, 2001 2 March, 2001 3 March, 2001
27 February, 2004 28 February, 2004 29 February, 2004 1 March, 2004 2 March, 2004
3-term for loop:
27 February, 2004 28 February, 2004 29 February, 2004 1 March, 2004 2 March, 2004
Enter a starting date (day month, year): 25 Dec 2018
The date is now 25 December, 2018, add how many days? 7
The date is now 1 January, 2019, add how many days?
```

# Operator Overloading Retrospective

- We created the enum class Month and overloaded
  - **<<** so we can just write “std::cout << month;”
  - **>>** so we can just write “std::cin >> month;”
  - Comparisons are provided for enum classes by default
  - **++** both pre- and post-incrementing
  - These are functions – an enum class cannot have methods
- We created the Date class (including Month) with custom and default constructors, and overloaded
  - **<<** so we can just write “std::cout << date;”
  - **>>** so we can just write “std::cin >> date;”
  - **==** and the 5 others so we can just write “date1 == date2”
  - **++** both pre- and post-, with **+** (both orders) and **+=** for adding to int
  - These are methods (except <<, >>, and int + Date) since Date is a class

# What Other Operators Can Be Overloaded?

- Virtually ALL of them!
- Those that cannot (an exhaustive list!):
  - Membership operator (`::`),
    also known as the scope resolution operator
  - Member access operator (`.`)
  - Member access through pointer to member operator (`.*`)
  - Ternary conditional operator (`?:`)
  - **`sizeof`** operator
- You do NOT need to know the above list for the exam
- DO know “most, not all, C++ operators can be overloaded”
- I’ll give you the method or function prototypes on the exam

# Other C++ Limitations

- You can define only existing operators  Know the first 3 for the exam!
  - You can't create your own custom operator such as $$ or @
- You can define operators only with the usual precedence, grouping, and number of operands
  - E.g., no unary <= (less than or equal) and no binary ! (not)
- You must specify at least one non-primitive type as an operand
  - int operator+(int,int);// error: you can’t overload built-in +
  - vector operator+(const vector&, const vector &);  // ok
- (Details) The overloads of operators && and || lose their short-circuit evaluation feature (the right-hand member *will* be evaluated)
- (Details, details) The overload of operator -> must either return a raw pointer, or return an object (by reference or by value) for which operator -> is in turn overloaded

# C++ Operator Recommendations

- Overload operators only with their conventional meaning
  - + should be addition, * should be multiplication, [ ] should be access, () should be a call, etc.
  - += should mean the same as = … +
  - You must determine what is “conventional” with the classes you define
- Don’t overload unless it offers significant readability advantages
  - Most classes need few operator overloads
  - << (and sometimes >>) are the most common
  - Collections (containers) usually also need [ ]

# What We Learned Today

- Overload operators like << and == by defining a *method* on the class or an (often *friend*) *function* for the class or enum
  - Look up the declaration* – it must match exactly!
    - https://en.wikipedia.org/wiki/Operators_in_C_and_C++
  - *Almost* any operator may be overloaded
  - You may NOT create your own operators
- Between typedefs and overloaded operators, we can craft our types to behave as desired!

* I will provide it on the exam, you needn't memorize any of them!

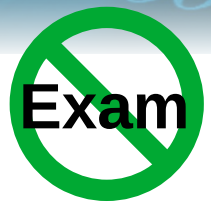
The End

# Overloading operator>> with Regular Expressions

- Streaming in the Month enum class *in the general form* is a challenge
  - But it would be *nice* to recognize several forms of month
    - “December”
    - “Dec”
    - “12”
    - An optional trailing comma
- A “regular expression” (regex) is often a great solution for recognizing input!

# Recognizing a Valid Month

- A regex is a compact expression of validity
- For January, we can “simply” write:




NOT on the exam
even as a bonus question

**[Jj] matches either a capital or lowercase J**

**[Jj]an matches either Jan or jan**

**Jan or January (caps or not) matches**

**A trailing , e.g., Jan, matches**

**([Jj]an(uary)?(,)?)|(0)?1(,)?**

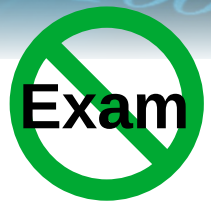
**All of the above is fine, OR...**

**An integer 1 or 01, with optional trailing ,**

Try https://www.regextester.com/97722 to interactively design a regex

# Regex Approach
# operator>> for a Month

- We could use a map or vector
  - Map: key would be Month and value is regex
  - Vector: key is int, value is {regex, Month}
- We'll do the vector first




NOT on the exam
even as a bonus question

**month_regex.cpp**

```
std::istream& operator>>(std::istream& is, Month& month) {
  class month_record{
    public:
      std::regex rx;
      Month month;
  };
  std::vector<month_record> month_records = {
    {std::regex{R"(([Jj]an(uary)?(,)?)|(0)?1(,)?)"},   Month::Jan},
    {std::regex{R"(([Ff]eb(uary)?(,)?)|(0)?2(,)?)"},   Month::Feb},
    {std::regex{R"(([Mm]ar()ch?(,)?)|(0)?3(,)?)"},     Month::Mar},
    {std::regex{R"(([Aa]pr(il)?(,)?)|(0)?4(,)?)"},     Month::Apr},
    {std::regex{R"(([Mm]ay(,)?)|(0)?5(,)?)"},          Month::May},
    {std::regex{R"(([Jj]un(e)?(,)?)|(0)?6(,)?)"},      Month::Jun},
    {std::regex{R"(([Jj]ul(y)?(,)?)|(0)?7(,)?)"},      Month::Jul},
    {std::regex{R"(([Aa]ug(ust)?(,)?)|(0)?8(,)?)"},    Month::Aug},
    {std::regex{R"(([Ss]ep(tember)?(,)?)|(0)?9(,)?)"}, Month::Sep},
    {std::regex{R"(([Oo]ct(ober)?(,)?)|10(,)?)"},      Month::Oct},
    {std::regex{R"(([Nn]ov(ember)?(,)?)|11(,)?)"},     Month::Nov},
    {std::regex{R"(([Dd]ec(ember)?(,)?)|12(,)?)"},     Month::Dec},
  };
```

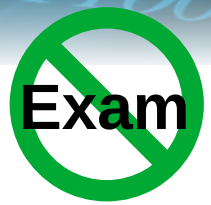
We can nest a class (or struct) in a function (!).
This is the value_type for our vector.

To create a raw string in which special chars (e.g., \n) are literals and never interpreted, start with `R"(` and end with `)"`

A raw strings is often used with a regex

## Regex Approach

# operator>> for a Month




NOT on the exam
even as a bonus question

- We need to stream in a string, then scan vector for a matching regex
  - If nothing matches, we throw an exception

**month_regex.cpp**

```
  std::string s;
  is >> s;

  for(auto mr : month_records) {
      if (std::regex_match(s, mr.rx)) {month = mr.month; s.clear(); break;}
  }

  if (!s.empty())
      throw std::runtime_error{"Invalid month: " + s};

  return is;
}
```

Stream in a string

Check all 12 regex for matches

Whatever comparison you prefer

On a match, set the month
and clear the string to signal success

If no match was found, throw an exception

# Regex Approach
# operator>> for a Month

- Here's the map version

**month_regex.cpp**

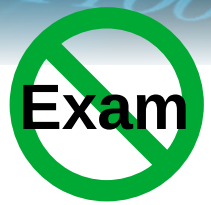
```
static const std::map<Month, std::regex> month_records {
    {Month::Jan, std::regex{R"(([Jj]an(uary)?(,)?)|(0)?1(,)?)"}},
    {Month::Feb, std::regex{R"(([Ff]eb(uary)?(,)?)|(0)?2(,)?)"}},
    {Month::Mar, std::regex{R"(([Mm]ar()ch?(,)?)|(0)?3(,)?)"}},
    {Month::Apr, std::regex{R"(([Aa]pr(il)?(,)?)|(0)?4(,)?)"}},
    {Month::May, std::regex{R"(([Mm]ay(,)?)|(0)?5(,)?)"}},
    {Month::Jun, std::regex{R"(([Jj]un(e)?(,)?)|(0)?6(,)?)"}},
    {Month::Jul, std::regex{R"(([Jj]ul(y)?(,)?)|(0)?7(,)?)"}},
    {Month::Aug, std::regex{R"(([Aa]ug(ust)?(,)?)|(0)?8(,)?)"}},
    {Month::Sep, std::regex{R"(([Ss]ep(tember)?(,)?)|(0)?9(,)?)"}},
    {Month::Oct, std::regex{R"(([Oo]ct(ober)?(,)?)|10(,)?)"}},
    {Month::Nov, std::regex{R"(([Nn]ov(ember)?(,)?)|11(,)?)"}},
    {Month::Dec, std::regex{R"(([Dd]ec(ember)?(,)?)|12(,)?)"}},
};
std::istream& operator>>(std::istream& is, Month& month) {
  std::string s; is >> s;
  for(auto& [mon, rx] : month_records) {
      if (std::regex_match(s, rx)) {
          month = mon; s.clear(); break;
      }
  }
  if (!s.empty()) throw std::runtime_error{"Invalid month: " + s};
  return is;
}
```

A std::regex can't be a map key unless you define an operator< function for it. This can be tricky, since the regex string is NOT kept in the object.

Unfortunately, we can't just do a lookup here since we also must do a regex_match with the key.
Map isn't as helpful for the regex version.

Regex Approach

# operator>> for a Month




NOT on the exam
even as a bonus question

- We need to stream in a string, then scan vector for a matching regex
  - If nothing matches, we throw an exception

month_regex.cpp

```
  std::string s;
  is >> s;

  for(auto mr : month_records) {
      if (std::regex_match(s, mr.rx)) {month = mr.month; s.clear(); break;}
  }

  if (!s.empty())
      throw std::runtime_error{"Invalid month: " + s};

  return is;
}
```

Stream in a string

Check all 12 regex for matches

Whatever comparison you prefer

On a match, set the month
and clear the string to signal success

If no match was found, throw an exception

# Polymorphic << and >>

- Polymorphism only works with methods
  - NOT constructors and NOT friends (neither may be virtual)
  - But << and >> can ONLY be defined as a friend!
- So how can we define << and >> for the superclass and polymorphically invoke it for the subclasses?

Challenge drawing by Clker-Free-Vector-Images per the Pixabay License

# Polymorphism with operator<<

```
#include <iostream>

class Base {
  public:
    Base(std::string s) : _s{s} { }
    friend std::ostream& operator<<(std::ostream& ost, const Base& base) {
        ost << base._s;        return ost;
    }
}
  private:
    std::string _s;
};
class Derived : public Base {
  public:
    Derived(std::string s, std::string t) : Base{s}, _t{t} { }
    friend std::ostream& operator<<(std::ostream& ost, const Derived& derived) {
        ost << static_cast<Base>(derived) << ' ' << derived._t;        return ost;
    }
  private:
    std::string _t;
};
int main() {
    Base b{"Hello"};
    std::cout << b << std::endl;
    Derived d{"Hi", "World"};
    std::cout << d << std::endl;
    Base& br = d;
    std::cout << br << std::endl;
}
```

Both Base and Derived overload the << operator.

But is operator<< polymorphic???

# Polymorphism with operator<<

```
#include <iostream>

class Base {
  public:
    Base(std::string s) : _s{s} { }
    friend std::ostream& operator<<(std::ostream& ost, const Base& base) {
        ost << base._s;        return ost;
    }
}
  private:
    std::string _s;
};
class D
  publi
    Der
    fri                                                    ived) {
                                                           turn ost;
    }
  priva
    std
};
int main() {
    Base b{"Hello"};
    std::cout << b << std::endl;
    Derived d{"Hi", "World"};
    std::cout << d << std::endl;
    Base& br = d;
    std::cout << br << std::endl;
}
```

Both Base and Derived overload the << operator.

```
ricegf@pluto:~/dev/cpp/201908/18$ ls
00.cpp  01.cpp  02.cpp  03.cpp  10.cpp  11.cpp  20.cpp  21.cpp  Makefile
ricegf@pluto:~/dev/cpp/201908/18$ make 20
g++ --std=c++17 20.cpp -o 20
ricegf@pluto:~/dev/cpp/201908/18$ ./20
Hello
Hi World
Hi
ricegf@pluto:~/dev/cpp/201908/18$
```

Nope!

# Methods *may* be Polymorphic Functions are NEVER Polymorphic

- Functions like operator<< cannot be polymorphic
  - They can't be overridden because they don't inherit
  - They may NEVER be declared virtual!
- What we need is a way to make operator<< rely on a polymorphic method
  - Revert to our tried and true `std::string to_string()` method
  - OR a `virtual std::ostream& print(std::ostream&)` method
  - EITHER could be called from `operator<<(std::ostream&, Base&)`

# Solving operator<<

```
#include <iostream>

class Base {
  public:
    Base(std::string s) : _s{s} { }
    virtual std::ostream& print(std::ostream& ost) const {
        ost << _s;          return ost;}
    friend std::ostream& operator<<(std::ostream& ost, const Base& base) {
        return base.print(ost);}
  private:
    std::string _s;
};
class Derived : public Base {
  public:
    Derived(std::string s, std::string t) : Base{s}, _t{t} { }
    virtual std::ostream& print(std::ostream& ost) const {
        ost << static_cast<Base>(*this) << ' ' << _t;     return ost;}
  private:
    std::string _t;
};
int main() {
    Base b{"Hello"};
    std::cout << b << std::endl;
    Derived d{"Hi", "World"};
    std::cout << d << std::endl;
    Base& br = d;
    std::cout << br << std::endl;
}
```

Operator<< now delegates to the polymorphic print method

Derived *no longer needs its own operator<<*;
Base's operator<< will delegate to Derive's print method *polymorphically*

Now operator<< *behaves* as if polymorphic by delegating to polymorphic print methods

# Solving operator<<

```
#include <iostream>

class Base {
  public:
    Base(std::string s) : _s{s} { }
    virtual std::ostream& print(std::ostream& ost) const {
        ost << _s;          return ost;}
    friend std::ostream& operator<<(std::ostream& ost, const Base& base) {
        return base.print(ost);}
  private:
    std::st
};
class Deriv
  public:
    Derivec
    virtua
        ost
  private:
    std::string _t;
};
int main() {
    Base b{"Hello"};
    std::cout << b << std::endl;
    Derived d{"Hi", "World"};
    std::cout << d << std::endl;
    Base& br = d;
    std::cout << br << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201908/18$ ls
00.cpp  01.cpp  02.cpp  03.cpp  10.cpp  11.cpp  20  20.cpp  21.cpp  Makefile
ricegf@pluto:~/dev/cpp/201908/18$ make 21
g++ --std=c++17 21.cpp -o 21
ricegf@pluto:~/dev/cpp/201908/18$ ./21
Hello
Hi World
Hi World
ricegf@pluto:~/dev/cpp/201908/18$
```

ethod

Derived *no longer needs its own operator<<*;
Base's operator<< will delegate to Derive's print method *polymorphically*

Now operator<< *behaves* as if polymorphic by delegating to polymorphic print methods