

Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

Université d'Ottawa | University of Ottawa



uOttawa

L'Université canadienne
Canada's university



uOttawa.ca

This Lecture

Just like int

- **Abstract Data Types**
 - Misc. topics
 - constexpr, Ch. 6.5.2
 - Abstract Data Types
 - Operator overloading, Ch. 14.1, 14.3, 14.7
 - Assignment operator, Ch. 13.2, 14.4
 - Conversions, Ch. 14.9
 - Example
 - Linear algebra

Constant Expressions

- **Constant expressions can be evaluated at compile time**
 - In C++11 functions can also be constant expressions
 - But only if the function can be evaluated at compile time
 - return type and all arguments must be literals
 - a function may be a constant expression depending on the argument

```
constexpr size_t arrlength( size_t _rows, size_t _cols ) {  
    return _rows * _cols + 10;  
}
```

Const Expression in C++14

- **C++14 extended what can be used inside constexpr**
 - declarations of local variables (but not static or uninitialized variables)
 - changing objects that were constructed with the const expression
 - using conditions (if, switch), and loops (for, while, do-while) but not goto

```
constexpr int factorial( int i ) {  
    int res = 1;  
    for ( int j=i; j > 0; --j ) res *= j;  
    return res;  
}
```

```
std::array<int, factorial(3)> A;
```

Needs to be known at
compile time

Operator Overloading

- Similar to function overloading
- Define new types for a C++ operator to work on
- **Examples: Overloaded assignment operator**
- **Some limitations**
 - Some operators can not be overloaded
 - . :: . * (Pointer-to-member)
 - ? : (conditional)
 - # ## (pre-processor strings)
- **All other operators can be overloaded!**

More Limitations

- Overloading the priority of operators is not possible
- No new operators can be created, e.g., no $!^!$
- Operators can not have default arguments
- For in class operators, first argument is always of the type for which the operator is defined

Example: Point2D Addition

```
class Point2D {  
...  
public:  
    Point2D add( const Point2D& _oPoint ) const;  
};  
  
Point2D Point2D::add( const Point2D& _oPoint ) const {  
    Point2D res;  
    res.d_x = d_x + _oPoint.d_x;  
    res.d_y = d_y + _oPoint.d_y;  
    return res;  
}  
  
Point2D a, b, c;  
c = a.add( b );
```

Example: Point2D Addition

```
class Point2D {  
    ...  
public:  
    Point2D operator+( const Point2D& _oPoint ) const;  
};  
  
Point2D Point2D::operator+( const Point2D& _oPoint ) const {  
    Point2D res;  
    res.d_x = d_x + _oPoint.d_x;  
    res.d_y = d_y + _oPoint.d_y;  
    return res;  
}  
  
Point2D a,b,c;  
c = a + b;
```

c = a.operator+(b);

Global Operator Overloading

- **Operators can be overloaded globally**
 - Same as any other global function
 - Need one extra argument compared to member operators
- **Example**
 - Insertion and extraction operator
- **Limitations**
 - Access modifiers apply (since not inside a class)
 - Can not globally overload
 - `=` `[]` `()` `->`
 - Additionally, same limitations than for class member operators

Example: Point2D Addition

```
Point2D operator+( const Point2D& p1, const Point2D& p2 );

Point2D operator+( const Point2D& p1, const Point2D& p2 ) {
    double x,y;
    x = p1.getX() + p2.getX();
    y = p1.getY() + p2.getY();
    return Point2D( x, y );
}

Point2D a,b,c;
c = a + b;
```

- **Same result than for member overloaded operator**
- **Why is this useful?**

Example: Addition of a double and a Point2d

- **Consider:**

```
Point2D a,b; b = 3.0 + a;
```

- Not possible with member overloaded operator since first argument is always an object of the class

- **Solution:**

```
Point2D operator+( double _val, const Point2D& _p );

Point2D operator+( double _val, const Point2D& _p ) {
    double x,y;
    x = _val + _p.getX();
    y = _val + _p.getY();
    return Point2D( x, y );
}

Point2D a, b;
b = 3.0 + a;
```

Example: Class for Timing Events

- **Time Operations**

- Difference between two times - -=
- Adding some extra time + ++ +=
- Subtracting some time -- -
- Rounding to the closest minute etc. ~
- Printing the time <<

TimeKeeper Class: Arithmetic Operations

```
class TimeKeeper {
    int d_seconds;
public:
    TimeKeeper(int sec=0) : d_seconds(sec) {}
    ... // Omitted
    TimeKeeper& operator+=(const TimeKeeper &rhs) {
        d_seconds+= rhs.d_seconds; return *this; }
    TimeKeeper& operator-=(const TimeKeeper &rhs) {
        d_seconds-= rhs.d_seconds; return *this; }
    const TimeKeeper operator~() const { TimeKeeper t(*this);
        if (t.sec()<30) t.d_seconds-= t.sec();
        else t.d_seconds+= 60-t.sec(); return t; }
    friend ostream& operator<<(ostream &lhs,
                                const TimeKeeper &rhs);
};
```

Remarks

- Prefix and Postfix operator are distinguished by a dummy (int) argument

```
TimeKeeper& operator++() { *this+= 1; return *this; }  
const TimeKeeper operator++(int) {  
    TimeKeeper tk(*this); ++(*this); return tk; }
```

- Addition and subtraction as global operators to enable automatic conversion from int (via constructor)

```
const TimeKeeper operator+(const TimeKeeper &lhs,  
                           const TimeKeeper &rhs) {  
    return TimeKeeper(lhs) += rhs; }  
const TimeKeeper operator-(const TimeKeeper &lhs,  
                           const TimeKeeper &rhs) {  
    return TimeKeeper(lhs) -= rhs; }
```

Conversion Operators

- **All (1-Argument until C++11) constructors act as conversion operators**
 - Creates a new temporary object then calls the copy constructor with the temporary
 - Copy constructors but also all other one-argument constructors
 - Sometimes intentional other times wasteful
- **explicit keyword**
 - Switches off the implicit conversion operation

Specific Conversion Operators

- **Conversion is a special operator**
 - Convert your type to another type including built-in types
 - Return value is determined from name
 - No return value (not even void)
 - No argument

```
operator type_name() { expression-list }
```

- C++11: Conversion operator can be made explicit
 - (must use cast except for conditions)

Improved Time Keeper Class

```
class TimeKeeper {
    int d_seconds;
public:
    // One argument constructor
    explicit TimeKeeper(int sec=0) : d_seconds(sec) {}
    // explicit conversion operator
    operator int() const { return d_seconds; }
    // Assignment operator from int
    TimeKeeper& operator=(int t) {d_seconds= t; return *this;}

    ... // Omitted

    // Aside: Increment can work directly on attributes
    TimeKeeper& operator++() { ++d_seconds; return *this; }
};
```

Next

Do more with less

- **Macros and Templates**
- **Textbook (Lippman): Chapters 2.9.2, 6.14, 16.1-16.3, 16.5**
 - Macros and the C++ preprocessor: debugging, conditional compilation
 - Templates: template functions and classes
 - Templates: type and non-type parameters
 - Template specialization