COMP6771
Advanced C++ Programming

4.1 – Operator Overloading

In this Lecture

Why?

- Operator overloads allow you to decrease your code complexity and utilise well defined semantics.

What?

- *Compile time polymorphism in existing operators*.
- Many different types of operator overloads.

Operator Overload Design

- Member operator overloads sometimes require **two** versions:
  - A non-const overload
  - A const overload
- A common example is `operator[]`:
  - The non-const overload is for setting values, e.g., `my_class[i] = 4;`
  - The const overload is for getting values
- Non-member operator overloads can *optionally* be friends.
  - If the operator overload can be implemented purely in terms of the class's **public** interface, it does not need to be a friend.
  - If it needs access to private or protected members, it should be a friend.
  - Non-member friend operator overloads should be **hidden friends,** i.e., defined inline in the class.
- In general, if an operator acts on a *particular* instance, it is a member function
  - Otherwise, it is a non-member function.

Motivating Example 1

- The last line is our best attempt to “Print the sum of two points”.
- **`print(std:cout, point::add(p, q));`**
- This is clumsy (and *ugly*!)
- We’d prefer to be able to write:
  - **`std:cout << p + q;`**

```
class point {
public:
  // implementation details...
  static point add(const point &p, const point &q);
  friend void print(std::ostream &, const point&);
private:
  int x_;
  int y_;
}

point point::add(const point &p, const point &q) {
   return point{p.x_ + q.x_, p.y_ + q.y_};
}

void print(std::ostream &os, const point &p) {
  os << “(“ << p.x_ << “,” << p.y_ << “)”;
}

point p = {1, 2}, q = {3, 4};

print(std:cout, point::add(p, q)); // EW!
```

Motivating Example 2

- Using **operator overloading**:
  - Allows us to reuse our intuition with operators to implement nicer semantics for our classes!
  - Gives us a common and simple interface to implement classes to behave like built-in types.

```
class point {
public:
  // implementation details...
  friend
  point operator+(const point &p, const point &q);

  friend
  void operator<<(std::ostream &os, const point&p);
private:
  int x_;
  int y_;
}

point operator+(const point &p, const point &q) {
   return point{p.x_ + q.x_, p.y_ + q.y_};
}

void operator<<(std::ostream &os, const point &p) {
  os << “(“ << p.x_ << “,” << p.y_ << “)”;
  return os;
}

point p = {1, 2}, q = {3, 4};

std:cout << p + q << std::endl; // excellent
```

Operator Overloading in C++

- C++ supports a rich set of operator overloads.
- All operator overloads must have at least one operand of its type.
- Advantages:
  - Readability & reuse of existing code semantics.
  - Flexible and easy to maintain different operations.
  - No verbosity required for simple operations.
- Disadvantages:
  - Easy to abuse and create an overload that is distinct from its original meaning.
  - Slower than built-in operators due to a function call vs. a CPU instruction.
  - Only create an overload if your type has a single, obvious meaning for an operation.

Operator Overload Canonical Implementations

Type	Operator(s)	Canonical Implementation
I/O	<<, >>	Non-member function
Arithmetic	+, -, *, /, %, + (unary), - (unary)	Non-member function
Bitwise	&, |, ^, ~, >>, <<	Non-member function
Compound Assignment	+=, -=, *=, /=, %=, ^=, |=, &=	Member function
Comparison	>, <, >=, <=, !=, ==, <=>	Member or non-member
Logical	&&, ||, !	Non-member function
Assignment	=	Member function
Subscript	[]	Member function
Increment / Decrement	++ / -- (pre), ++ / -- (post)	Member function
Member Access & Dereference	->, *	Member function
Type Conversions	static_cast<type>	Member function
Call Operator	()	Member function

Overload: I/O

- Scope to overload for different types of output and input streams.
- Not member functions!
  - Why?
- `operator>>` takes a non-const reference to a point
  - Why?
- Returns a reference to an iostream.
  - Why?

```
struct point {
  int x;
  int y;

  friend
  std::ostream &
  operator<<(std::ostream &os, const point &p) {
     os << "(" << p.x << "," << p.y << ")";
     return os;
  }

  friend
  std::istream &
  operator>>(std::istream& is, point& p) {
     is >> p.x >> p.y;
     return is;
  }
};

int main() {
  point p = {1, 2};
  std::cout << p << std::endl;
}
```

Overload: Arithmetic

- Classes that model mathematical objects often have the arithmetic operators defined.
- If you define one, it is best to define all of them.
- All operators return a new object.

```
class vec2 {
public:
  // other implementation details

  friend
  vec2 operator+(const vec2 &u, const vec2 &v) {
    return vec2{u.x_ + v.x_, u.y_, v.y_};
  }

  friend
  vec2 operator-(const vec2 &u, const vec2 &v) {
    return vec2{u.x_ - v.x_, u.y_ - v.y_};
  }

  // etc. for the other operators

private:
  double x_;
  double y_;
};
```

Overload: Bitwise

- The bitwise operators can be overloaded.
- Useful for making enum classes act like bit-flags.

```
enum class flag : int {
  O_NONE  = 0b000,
  O_READ  = 0b001,
  O_WRITE = 0b010,
  O_RW    = 0b100
};

flag
operator|(const flag &f1, const flag &f2) {
  int f1i = static_cast<int>(f1);
  int f2i = static_cast<int>(f2);

  return static_cast<flag>(f1i | f2i);
}


flag
operator&(const flag &f1, const flag &f2) {
  int f1i = static_cast<int>(f1);
  int f2i = static_cast<int>(f2);

  return static_cast<flag>(f1i & f2i);
}

flag my_flags = flag::O_READ | flag::O_WRITE;
```

Overload: Compound Assignment

- Each class can have any number of **`operator+=`** operators, but there can only be one `operator+=X` (where `X` is a type).
  - That's why in this case we have two multiplier compound assignment operators.

```
class vec2 {
public:
  // other implementation details

  vec2 &operator+=(const vec2 &v) {
    x_ += v.x_;
    y_ += v.y_;
    return *this;
  }

  // how would these be implemented?
  vec2 &operator-=(const vec2 &v);
  vec2 &operator*=(double scale);
  vec2 &operator*=(int scale);
  vec2 &operator/=(double scale);

private:
  double x_;
  double y_;
};
```

Overload: Comparisons

- Most types should implement at least the equality (==, !=) operators.
- If the type has an *ordering*, then it should implement the relational (<, <=, >, >=) operators too.
- This is largely a hassle.
  - Minimum functions to write are `operator==` and `operator<`.
  - The other four operators can be written in terms of these two.
  - C++ introduced the **spaceship** operator to solve this.

```
class vec2 {
public:
  // other implementation details

  double x() const { return x_; }
  double y() const { return y_; }

private:
  double x_;
  double y_;
};

bool operator<(const vec2 &l, const vec2 &r);
bool operator>(const vec2 &l, const vec2 &r);
bool operator<=(const vec2 &l, const vec2 &r);
bool operator<=(const vec2 &l, const vec2 &r);
bool operator==(const vec2 &l, const vec2 &r);
bool operator!=(const vec2 &l, const vec2 &r);
```

Overload: Spaceship Operator

- New in C++20: three-way comparison with operator<=>.
  - If `a < b`, `(a <=> b) < 0`
  - If `a > b`, `(a <=> b) > 0`
  - If a is equivalent or equal to b, `(a <=> b) == 0`
- `a <=> b` returns one of three kinds of *orderings*:
  - `std::strong_ordering`
  - `std::weak_ordering`
  - `std::partial_ordering`
- All orderings support "less than" and "greater than".
- Only std::strong_ordering supports equality (if `a == b` and `b == c`, `a == c`).
- `std::weak_ordering` and `std::partial_ordering` support "equivalence" (a is neither less than nor greater than b, but not equal).
- Only `std::partial_ordering` supports incomparable values (`a <=> b` is always false).

Ordering	Equivalent values are…	Incomparable values are…
std::strong_ordering	Indistinguishable	Disallowed
std::weak_ordering	Distinguishable	Disallowed
std::partial_ordering	Distinguishable	Allowed

Comparison Category	**Use Cases**

`std::partial_ordering::less`
`std::partial_ordering::equivalent`
`std::partial_ordering::greater`
`std::partial_ordering::unordered`
	• Floating-point numbers
	• Complex numbers
	• 2D points

`std::weak_ordering::less`
`std::weak_ordering::equivalent`
`std::weak_ordering::greater`
	• Case insensitive strings

`std::strong_ordering::less`
`std::strong_ordering::equal`
`std::strong_ordering::greater`
	• Integral types
	• Strings
	• 1D arrays

Spaceship Operator Example

```
#include <compare> // needed for std::partial_ordering

class point {
public:
  // other implementation details

  friend std::partial_ordering
  operator<=>(const point &p1, const point &p2) {
    auto x_ord = p1.x_ <=> p2.x_;
    auto y_ord = p1.y_ <=> p2.y_;

    return x_ord == y_ord ? x_ord : std::partial_ordering::unordered;
  }

private:
    int x_;
    int y_;
};
```

Default Comparisons (since C++20)

- `operator<=>` is called whenever objects are compared with a relational operator.
- `operator==` is called when objects are compared with an equality operator.
- It is possible to default these member functions since C++20.
  - Default behaviour compares members in declaration order using the appropriate operator.

```
struct point2 {
  double x;
  double y;

  auto // could also be a friend
  operator<=>(const point2 &) const = default;

  // ALL other comparison operators
  // automatically synthesised
};

struct vec2 {
  double x;
  double y;

  bool // must be bool
  operator==(const vec2 &) const = default;

  // only operator==, operator!= synthesised
};
```

Overload: Logical Operators

- Logical AND, OR, and NOT can be overloaded.
- They will **lose** their short-circuit behaviour if overloaded.
- Not a common overload.

```
class Bool { // a boxed bool
public:
  // other implementation details

  friend bool
  operator&&(Bool a, Bool b) {
    return Bool{a.b_ && b.b_};
  }

  // similar for the other operators
  friend bool operator||(Bool a, Bool b);
  friend bool operator!(Bool a, Bool b);

private:
  bool b_;
};
```

Overload: Assignment

- Assignment operator is one of the special member functions.
- Need to be careful to avoid self-assignment.
- Can be defaulted or deleted.
  - Default behaviour is to go through all members in declaration order and try to assign them.
- **Two** kinds of assignment operators
  - Copy assignment
  - Move assignment (will discuss later)

```
struct vec2 {
  int x;
  int y;

  // canonical copy assignment signature
  // defaulted: the compiler generates it
  // will try to copy assign v.x to x
  // and v.y to y.
  vec2 &operator=(const vec2 &v) {
    if (&v != this) {
       // do this copy
    }
  }

  // canonical move assignment operator
  // don't need to worry about this yet.
  vec2 &operator(vec2 &&v) = delete;
};
```

Overload: Subscript

- Usually only defined on indexable containers.
- Need two overloads for get & set.
- During development, asserts can be used for bounds checking:
  - In other containers (e.g. vector), invalid index access is undefined behaviour.
  - Usually an explicit crash is better than undefined behaviour.
  - Asserts are stripped out of optimised builds, so no performance penalty.
- The getter version can either return by value or const reference.
  - If copying is expensive, const reference is preferred.

```
#include <cassert>

struct vec2 {
  int x;
  int y;

  int &operator[](int n) {
    assert(n == 0 || n == 1);
    return n == 0 ? x : y;
  }

  int  operator[](int n) const {
    assert(n == 0 || n == 1);
    return n == 0 ? x : y;
  }
};
```

Overload: Increment & Decrement

- Prefix:
  - ++x
  - --x
  - Returns a reference
- Postfix:
  - x++
  - x—
  - Returns a copy
- Need two overloads for pre- vs. post-fix.
  - Use an anonymous `int` variable in the **postfix** overload.
  - It is only used for overload resolution.
- Performance: prefix > postfix.

```
class tick {
public:
  // other implementation details

  tick &operator++() {
    cnt_++;
    return *this;
  }

  tick operator++(int) {
    auto self = *this;
    ++*this;
    return self;
  }

  // similar for decrement
  tick &operator—();
  tick operator—(int);

private:
  int cnt_;
};
```

Overload: Member Access & Dereference

- Classes exhibit pointer-like behaviour when -> and * are overloaded
- For -> to work it must return a pointer to a class type or an object of a class type that defines its own -> operator
- Useful for making “smart” pointers.

```
struct point { int x; int y; };

class point_ptr {
public:
  point_ptr(int x, int y)
  : ptr_{new point{x, y}} {}

  point &operator*() const {
    return *ptr;
  }

  point *operator->() const {
    return ptr_;
  }

  ~point_ptr() { delete ptr_; }

private:
  point *ptr_;
};
```

Overload: Type Conversions

- Define how a class type is converted into another type.
- Syntax for these operators look similar to constructors.
- Virtually always const-qualified.

```
struct centimeter { double cnt; };

struct inch {
  double cnt;

  operator centimeter() const {
    return centimeter{cnt * 2.45};
  }
};
```

Overload: Call Operator

- Make an instance of a class type callable
  - i.e., create *functors* (function objects)
- Can have many different overloads of operator() so long as they have different parameters.

```
struct comparator {
  bool operator()(int l, int r) {
    return l <=> r;
  }

  bool operator()(char l, char r) {
    return l <=> r;
  }
};

auto ints = std::vector<int>{3, 1, 2};
auto chars = std::vector<char>{‘c’, ‘a’, ‘b’};
auto cmp = comparator{};

std::sort(ints.begin(), ints.end(), cmp);
std::sort(chars.begin(), chars.end(), cmp);
```

Operator Piggybacking

You'll notice that *many* operator overloads can be written in terms of other ones. The below table which operators are written in terms of which others.

Operator…	**Can be written in terms of…**
`operator@` (where `@` is one of the arithmetic or bitwise operators)	`operator@=`
`operator!=`	`operator==`
`operator$` (where `$` is one of `<=`, `>`, `>=`)	`operator<` and `operator==`
`operator++(int)`	`operator++()`
`operator—(int)`	`operator--()`
`operator->`	`operator*`

Miscellaneous Operators

- There are some operators that are rarely overloaded:
  - Allocation function (`operator new / operator new[]`)
  - Deallocation function (`operator delete / operator delete[]`)
  - The address-of operator (`operator&`)
  - The member-access-through-pointer operator (`operator->*`)
  - The comma operator (`operator,`)
  - User-defined literals (since C++11) (`operator""`)
- There are also some operators that cannot be overloaded.
  - The ternary operator (`operator?`)
  - The member-access operator (`operator.`)
    - If this were possible, one could create “smart references”.

Feedback (stop recording)