COMP6771
Advanced C++ Programming

Week 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

Start with an example

```cpp
 1 #include <iostream>
 2 
 3 class point {
 4 public:
 5         point(int x, int y)
 6         : x_{x}
 7         , y_{y} {};
 8         [[nodiscard]] int const x() const {
 9                 return this->x_; }
10         [[nodiscard]] int const y() const {
11                 return this->y_; }
12         static point add(point const& p1, point const& p2);
13 
14 private:
15         int x_;
16         int y_;
17 };
18 
19 void print(std::ostream& os, point const& p) {
20         os << "(" << p.x() << "," << p.y() << ")";
21 }
22 
23 point point::add(point const& p1, point const& p2) {
24         return point{p1.x() + p2.x(), p1.y() + p2.y()};
25 }
26 
27 auto main() -> int {
28         point p1{1, 2};
29         point p2{2, 3};
30         print(std::cout, point::add(p1, p2));
31         std::cout << "\n";
32 }
```

lecture-3/demo401-point1.cpp

Line 30 is our best attempt to "Add two points together and print them"

print(std::cout, point::add(p1, p2));

This is clumsy and ugly. We'd much prefer to have a semantic like this

std::cout << p1 + p2;

Start with an example

```cpp
#include <iostream>

class point {
public:
        point(int x, int y)
        : x_{x}
        , y_{y} {};
        friend point operator+(point const& lhs,
       point const& rhs);
        friend std::ostream& operator<<(std::ostream& os,
       point const& p);

private:
        int x_;
        int y_;
};

point operator+(point const& lhs, point const& rhs) {
        return point(lhs.x_ + rhs.x_, lhs.y_ + rhs.y_);
}

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

auto main() -> int {
        point p1{1, 2};
        point p2{2, 3};
        std::cout << p1 + p2 << "\n";
}
```

lecture-3/demo402-point2.cpp

Using operator overloading:

- Allows us to use currently understood semantics (all of the operators!)
- Gives us a common and simple interface to define class methods

Operator Overloading

- C++ supports a rich set of operator overloads
- All operator overloads must have at least one operand of its type
- Advantages:
  - Readability & Reuse existing code semantics
  - save memory space, consistency, and readability: speeds
  - Flexible and easy to maintain, different operations
  - No verbosity required for simple operations
- Disadvantages:
  - Lack of context on operations
- Only create an overload if your type has a single, obvious meaning to an operator

Friends

- A class may declare friend functions or classes
    - Those functions / classes are non-member functions that may access private parts of the class
    - This is, in general, a bad idea, but there are a few cases where it may be required
        - Nonmember operator overloads (will be discussing soon)
        - Related classes
            - A Window class might have WindowManager as a friend
            - A TreeNode class might have a Tree as a friend
            - Container could have iterator_t<Container> as a friend
                - Though a nested class may be more appropriate
    - Use friends when:
        - The data should not be available to everyone
        - There is a piece of code very related to this particular class

In general we prefer to define friends directly in the class they relate to

Operator Overload Design

Type	Operator(s)	Member / friend
I/O	<<, >>	friend
Arithmetic	+, -, *, /	friend
Relational, Equality	>, <, >=, <=, ==, !=	friend
Assignment	=	member (non-const)
Compound assignment	+=, -=, *=, /=	member (non-const)
Subscript	[]	member (const and non-const)
Increment/Decrement	++, --	member (non-const)
Arrow, Deference	->, *	member (const and non-const)
Call	()	member

- Use friends when the operation is called without any particular instance
  - Even if they don't require access to private details
- Use members when the operation is called in the context of a particular instance

Overload: I/O

```cpp
#include <istream>
#include <ostream>

class point {
public:
	point(int x, int y)
	: x_{x}
	, y_{y} {};
	friend std::ostream& operator<<(std::ostream& os, const point& type);
	friend std::istream& operator>>(std::istream& is, point& type);

private:
	int x_;
	int y_;
};

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

std::istream& operator>>(std::istream& is, point& p) {
	// To be done in tutorials
}

auto main() -> int {
	point p(1, 2);
	std::cout << p << '\n';
}
```

lecture-3/demo403-io.cpp

- Equivalent to .toString() method in Java
- Scope to overload for different types of output and input streams
- Why not member function?
- Operands ?
- ostream& return type ?

(std::cout << point) << '\n';

Overload: Compound assignment

```cpp
class point {
public:
	point(int x, int y)
	: x_{x}
	, y_{y} {};
	point& operator+=(point const& p);
	point& operator-=(point const& p);
	point& operator*=(point const& p);
	point& operator/=(point const& p);
	point& operator*=(int i);

private:
	int x_;
	int y_;
};

point& point::operator+=(point const& p) {
	x_ += p.x_;
	y_ += p.y_;
	return *this;
}

point& operator+=(point const& p) { /* what do we put here? */}
point& operator-=(point const& p) { /* what do we put here? */}
point& operator*=(point const& p) { /* what do we put here? */}
point& operator/=(point const& p) { /* what do we put here? */}
point& operator*=(int i) { /* what do we put here? */}
```

lecture-3/demo404-compassign.cpp

- Sometimes particular methods might not have any real meaning, and they should be omitted (in this case, what does dividing two points together mean).
- 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

Operator pairings

Many operators should be grouped together. This table should help you work out which are the minimal set of operators to overload for any particular operator.

If you overload	Then you should also overload		
`operator OP=(T, U)`	`operator OP(T, U)`		
`operator+(T, U)`	`operator+(U, T)`		
`operator-(T, U)`	`operator+(T, U)`	`operator+(T)`	`operator-(T)`
`operator/(T, U)`	`operator*(T, U)`		
`operator%(T, U)`	`operator/(T, U)`		
`operator++()`	`operator++(int)`		
`operator--()`	`operator++()`	`operator--(int)`	
`operator->()`	`operator*()`		

Overload: Relational & Equality

```cpp
#include <iostream>

class point {
public:
	point(int x, int y)
	: x_{x}
	, y_{y} {}
	// hidden friend - preferred
	friend bool operator==(point const& p1, point const& p2) {
		return p1.x_ == p2.x_ and p1.y_ == p2.y_;
		// return std::tie(p1.x_, p1.y_) == std::tie(p2.x_, p2.y_);
	}
	friend bool operator!=(point const& p1, point const& p2) {
		return not (p1 == p2);
	}
	friend bool operator<(point const& p1, point const& p2) {
		return p1.x_ < p2.x_ and p1.y_ < p2.y_;
	}
	friend bool operator>(point const& p1, point const& p2) {
		return p2 < p1;
	}
	friend bool operator<=(point const& p1, point const& p2) {
		return not (p2 < p1);
	}
	friend bool operator>=(point const& p1, point const& p2) {
		return not (p1 < p2);
	}

private:
	int x_;
	int y_;
};

auto main() -> int {
	auto const p2 = point{1, 2};
	auto const p1 = point{1, 2};
	std::cout << "p1 == p2 " << (p1 == p2) << '\n';
	std::cout << "p1 != p2 " << (p1 != p2) << '\n';
	std::cout << "p1 < p2 " << (p1 < p2) << '\n';
	std::cout << "p1 > p2 " << (p1 > p2) << '\n';
	std::cout << "p1 <= p2 " << (p1 <= p2) << '\n';
	std::cout << "p1 >= p2 " << (p1 >= p2) << '\n';
}
```

lecture-3/demo405-relation1.cpp

- Do we want all of these?
- We're able to "piggyback" off previous definitions
- Check out the spaceship operator

Overload: Assignment

```cpp
#include <istream>

class point {
public:
        point(int x, int y)
        : x_{x}
        , y_{y} {};
        point& operator=(point const& p);

private:
        int x_;
        int y_;
};

point& point::operator=(point const& p) {
        x_ = p.x_;
        y_ = p.y_;
        return *this;
}
```

lecture-3/demo406-assign.h

- Similar to compound assignment
- When should we doing ? ob1=ob2;
  - Deep Copy
- Operands
- must be overload by non-static

Overload: Subscript

```cpp
#include <cassert>

class point {
public:
	point(int x, int y)
	: x_{x}
	, y_{y} {};
	int& operator[](int i) {
		assert(i == 0 or i == 1);
		return i == 0 ? x_ : y_;
	}
	int operator[](int i) const {
		assert(i == 0 or i == 1);
		return i == 0 ? x_ : y_;
	}

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

lecture-3/demo407-subscript.h

- Usually only defined on indexable containers
- Different operator for get/set
- Asserts are the right approach here as preconditions:
  - 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
  - can be used to check bound cases
  - non-static just like () -> =

Overload: Increment/Decrement

```cpp
// RoadPosition.h:
class RoadPosition {
  public:
    RoadPosition(int km) : km_from_sydney_(km) {}
    RoadPosition& operator++();      // prefix
    // This is *always* an int, no
    // matter your type.
    RoadPosition operator++(int);   // postfix
    void tick();
    int km() { return km_from_sydney_; }

  private:
    void tick_();
    int km_from_sydney_;
};

// RoadPosition.cpp:
#include <iostream>
RoadPosition& RoadPosition::operator++() {
  this->tick_();
  return *this;
}
RoadPosition RoadPosition::operator++(int) {
  RoadPosition rp = *this;
  this->tick_();
  return rp;
}
void RoadPosition::tick_() {
  ++(this->km_from_sydney_);
}
```

lecture-3/demo408-incdec.h

- prefix: ++x, --x, returns lvalue reference
  - Discussed more in week 5
- postfix: x++, x--, returns rvalue
  - Discussed more in week 5
- Performance: prefix > postfix
- Different operator for get/set
- Postfix operator takes in an int
  - This is not to be used
  - It is only for function matching
  - Don't name the variable

```cpp
auto main() -> int {
  auto rp = RoadPosition(5);
  std::cout << rp.km() << '\n';
  auto val1 = (rp++).km();
  auto val2 = (++rp).km();
  std::cout << val1 << '\n';
  std::cout << val2 << '\n';
}
```

lecture-3/demo408-incdec.cpp

Overload: Arrow & Dereferencing

```cpp
#include <iostream>
class stringptr {
public:
	explicit stringptr(std::string const& s)
	: ptr_{new std::string(s)} {}
	~stringptr() {
		delete ptr_;
	}
	std::string* operator->() const {
		return ptr_;
	}
	std::string& operator*() const {
		return *ptr_;
	}

private:
	std::string* ptr_;
};

auto main() -> int {
	auto p = stringptr("smart pointer");
	std::cout << *p << '\n';
	std::cout << p->size() << '\n';
}
```

lecture-3/demo409-arrow.cpp

- **This content will feature heavily in week 5**
- Classes exhibit pointer-like behaviour when -> is 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

Overload: Type Conversion

```cpp
#include <vector>

class point {
public:
	point(int x, int y)
	: x_(x)
	, y_(y) {}
	explicit operator std::vector<int>() {  // usd-aud
		std::vector<int> vec;
		vec.push_back(x_);
		vec.push_back(y_);
		return vec;
	}

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

lecture-3/demo410-type.h

```cpp
#include <iostream>
#include <vector>
int main() {
	auto p = point(1, 2);
	auto vec = static_cast<std::vector<int>>(p);
	std::cout << vec[0] << '\n';
	std::cout << vec[1] << '\n';
}
```

lecture-3/demo410-type.cpp

- Many other operator overloads
  - Full list here: https://en.cppreference.com/w/cpp/language/operators
  - Example: <type> overload

Overload: New Function Syntax

```cpp
#include <iostream>
class stringptr {
public:
        explicit stringptr(std::string const& s)
        : ptr_{new std::string(s)} {}
        ~stringptr() {
                delete ptr_;
        }
        auto operator->() const -> std::string* {
                return ptr_;
        }
        auto operator*() const -> std::string& {
                return *ptr_;
        }

private:
        std::string* ptr_;
};

void * operator new(size_t size){
void *pointer;
cout<<"overloaded"
}
void operator delete(void *pointer)
{
free(pointer);
}
auto main() -> int {
        auto p = stringptr("smart pointer");
        std::cout << *p << '\n';
        std::cout << p->size() << '\n';
```

lecture-3/demo411-syntax.cpp

- We are able to use the new function syntax on our operator overloads as well
- add flexibility for heap allocation-layer
- static member function so no access to this pointer: auto static
- dont forget to overload array version
  - int *ovl= new int[5];
- overload new-delete
- multiple overload del/new operator
- void * operator new (size_t size){}

Overload: Spaceship Operator

```cpp
#include <compare>
#include <iostream>

class point {
public:
    point(int x, int y)
    : x_{x}
    , y_{y} {}

    // hidden friend - preferred
    // return type deduced as std::strong_ordering
    friend auto operator<=>(point p1, point p2) = default;

private:
    int x_;
    int y_;
};

auto main() -> int {
    auto const p2 = point{1, 2};
    auto const p1 = point{1, 2};
    std::cout << "p1 == p2 " << (p1 == p2) << '\n';
    std::cout << "p1 != p2 " << (p1 != p2) << '\n';
    std::cout << "p1 < p2 " << (p1 < p2) << '\n';
    std::cout << "p1 > p2 " << (p1 > p2) << '\n';
    std::cout << "p1 <= p2 " << (p1 <= p2) << '\n';
    std::cout << "p1 >= p2 " << (p1 >= p2) << '\n';
}
```

```cpp
#include <compare>
#include <iostream>

class point {
public:
    point(double x, double y)
    : x_{x}
    , y_{y} {}

    // hidden friend - preferred
    // return type deduced as std::partial_ordering
    friend auto operator<=>(point p1, point p2) = default;

private:
    double x_;
    double y_;
};

auto main() -> int {
    auto const p2 = point{1.0, 2.0};
    auto const p1 = point{1.0, 2.0};
    std::cout << "p1 == p2 " << (p1 == p2) << '\n';
    std::cout << "p1 != p2 " << (p1 != p2) << '\n';
    std::cout << "p1 < p2 " << (p1 < p2) << '\n';
    std::cout << "p1 > p2 " << (p1 > p2) << '\n';
    std::cout << "p1 <= p2 " << (p1 <= p2) << '\n';
    std::cout << "p1 >= p2 " << (p1 >= p2) << '\n';
}
```

lecture-3/demo405-relation2.cpp

Overload: Spaceship Operator

```
// For int-based point
auto const ordering = (p1 <=> p2) == std::strong_ordering::equal;
std::cout << "p1 <=> p2 yields equal " << ordering << '\n';
```

```
// For double-based point
auto const ordering = (p1 <=> p2) == std::partial_ordering::equivalent;
std::cout << "p1 <=> p2 yields equivalent " << ordering << '\n';
```

Overload: Spaceship Operator

```
#include <compare>
```

Example types

```
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
```

- Integers
- std::string

Overload: Spaceship Operator

```cpp
#include <compare>
#include <iostream>

class point {
public:
    point(int x, int y)
    : x_{x}
    , y_{y} {}

    friend auto operator==(point, point) -> bool = default;

    friend auto operator<=>(point const p1, point const p2) -> std::partial_ordering {
        auto const x_result = p1.x_ <=> p2.x_;
        auto const y_result = p1.y_ <=> p2.y_;
        return x_result == y_result ? x_result
                                    : std::partial_ordering::unordered;
    }

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

lecture-3/demo405-relation2.cpp

Feedback