# COMP6771 Week 4.1

## Operator Overloading

# Start with an example

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

- Line 26 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>
#include <ostream>

class Point {
  public:
    Point(int x, int y) : x_{x}, y_{y} {};
    const int& x() const { return this->x_; };
    const int& y() const { return this->y_; };
    static Point add(const Point& p1, const Point& p2);
  private:
    int x_;
    int y_;
};

void print(std::ostream& os, const Point& p) {
  os << "(" << p.x() << "," << p.y() << ")";
}

Point Point::add(const Point& p1, const Point& p2) {
  return Point{p1.x() + p2.x(), p1.y() + p2.y()};
}

int main() {
  Point p1{1, 2};
  Point p2{2, 3};
  print(std::cout, Point::add(p1, p2));
  std::cout << "\n";
}
```

```cpp
#include <iostream>
#include <ostream>

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

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

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

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

# Operator Overloading

- C++ supports a rich set of operator overloads
- All operator overloads must have at least one operand of its type
- Advantages:
  - Reuse existing code semantics
  - 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

# 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 (both) |
| Increment/Decrement | ++, -- | member (non-const) |
| Arrow, Deference | ->, * | member (both) |
| Call | () | member |

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

# Overload: I/O

```
1  // Point.h:
2  #include <ostream>
3  #include <istream>
4  class Point {
5    public:
6      Point(int x, int y) : x_{x}, y_{y} {};
7      friend std::ostream& operator<<(std::ostream& os, const Point& type);
8      friend std::istream& operator>>(std::istream& is, Point& type);
9
10   private:
11     int x_;
12     int y_;
13 };
14
15 // Point.cpp:
16 #include <ostream>
17 #include <istream>
18 #include <oistream>
19 std::ostream& operator<<(std::ostream& os, const Point& p) {
20   os << "(" << p.x_ << "," << p.y_ << ")";
21   return os;
22 }
23 std::istream& operator>>(std::istream& is, Point& type) {
24   // To be done in tutorials
25 }
26
27 int main() {
28   Point p{1,2};
29   std::cout << p << '\n';
30 }
```

# Overload: Compound assignment

```
1  // Point.h:
2  class Point {
3    public:
4      Point& operator+=(const Point& p);
5      Point& operator-=(const Point& p);
6      Point& operator*=(const Point& p);
7      Point& operator/=(const Point& p);
8      Point& operator*=(const int& i);
9
10   private:
11     int x_;
12     int y_;
13 };
14
15 // Point.cpp:
16 Point& Point::operator+=(const Point& p) {
17   this->x_ += p.x_;
18   this->y_ += p.y_;
19   return *this;
20 }
21 Point& Point::operator-=(const Point& p) { /* Should we do this one? */ }
22 Point& Point::operator*=(const Point& p) { /* Should we do this one? */ }
23 Point& Point::operator/=(const Point& p) { /* Should we do this one? */ }
24 Point& Point::operator*=(const int& p) { /* Should we do this one? */ }
```

# Overload: Relational & Equality

```
1  // Point.h:
2  class Point {
3    public:
4      // hidden friend - preferred
5      friend bool operator==(const Point& p1, const Point& p2) {
6        return p1.x_ == p2.x_ && p1.y_ == p2.y_;
7        // return std::tie(p1.x_, p1.y_) == std::tie(p2.x_, p2.y_);
8      }
9      friend bool operator!=(const Point& p1, const Point& p2) {
10       return !(p1 == p2);
11     }
12     friend bool operator<(const Point& p1, const Point& p2) {
13       // Do we want this? Alternatives?
14     }
15     friend bool operator<=(const Point& p1, const Point& p2);
16     friend bool operator>(const Point& p1, const Point& p2);
17     friend bool operator>=(const Point& p1, const Point& p2);
18
19   private:
20     int x_;
21     int y_;
22 };
```

# Overload: Assignment

```
1  // Point.h:
2  #include <istream>
3  class Point {
4    public:
5      Point& operator=(const Point& p);
6      Point& operator=(std::istream &is);
7
8    private:
9      int x_;
10     int y_;
11 };
12
13 // Point.cpp:
14 #include <istream>
15 Point& Point::operator=(const Point& p) {
16   this->x_ = p.x_;
17   this->y_ = p.y_;
18   return *this;
19 }
20 Point& Point::operator=(std::istream &is) {
21   // etc
22 }
```

# Overload: Subscript

- Usually only defined on indexable containers
- Different operator for get/set

```
1  // Point.h:
2  class Point {
3    public:
4      int& operator[](int i);       // setting via []
5      int  operator[](int i) const; // getting via []
6
7    private:
8      int x_;
9      int y_;
10 };
11
12 // Point.cpp:
13 #include <cassert>
14 int& Point::operator[](int i) {
15   assert(i == 0 || i == 1);
16   if (i == 0) return this->x_;
17   else return this->y_;
18 };
19 int Point::operator[](int i) const {
20   assert(i == 0 || i == 1);
21   if (i == 0) return this->x_;
22   else return this->y_;
23 };
```

- 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 optimisation builds

# Overload: Increment/Decrement

```
1  // RoadPosition.h:
2  class RoadPosition {
3    public:
4      RoadPosition(int km) : km_from_sydney_(km) {}
5      RoadPosition& operator++();      // prefix
6      // This is *always* an int, no
7      // matter your type.
8      RoadPosition operator++(int);   // postfix
9      void tick();
10     int km() { return km_from_sydney_; }
11
12   private:
13     void tick_();
14     int km_from_sydney_;
15 };
16
17 // RoadPosition.cpp:
18 #include <iostream>
19 RoadPosition& RoadPosition::operator++() {
20   this->tick_();
21   return *this;
22 }
23 RoadPosition RoadPosition::operator++(int) {
24   RoadPosition rp = *this;
25   this->tick_();
26   return rp;
27 }
28 void RoadPosition::tick_() {
29   ++(this->km_from_sydney_);
30 }
```

- prefix: ++x, --x, returns lvalue reference
- postfix: x++, x--, returns rvalue
- 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

```
1 int main() {
2   RoadPosition rp{5};
3   std::cout << rp.km() << '\n';
4   int val1 = (rp++).km();
5   int val2 = (++rp).km();
6   std::cout << val1 << '\n';
7   std::cout << val2 << '\n';
8 }
```

# Overload: Arrow & Dereferencing

```
1  #include <iostream>
2  class StringPtr {
3    public:
4      StringPtr(std::string *p) : ptr{p} { }
5      ~StringPtr() { delete ptr; }
6      std::string* operator->() { return ptr; }
7      std::string& operator*() { return *ptr; }
8    private:
9      std::string *ptr;
10 };
11
12 int main() {
13   std::string *ps = new std::string{"smart pointer"};
14   StringPtr p{ps};
15   std::cout << *p << std::endl;
16   std::cout << p->size() << std::endl;
17 }
```

- 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
- Interesting example: std::optional

# Overload: Other

```cpp
// Point.h:
#include <vector>
class Point {
  public:
    Point(int x, int y) : x_(x), y_(y) {}
    operator std::vector<int>() {
      std::vector<int> vec;
      vec.push_back(x_);
      vec.push_back(y_);
      return vec;
    }

  private:
    int x_;
    int y_;
};

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

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