CS100 Recitation 11

GKxx

返回多个值

想返回 $(\min, \max)$ 两个值，怎么办?

```
??? getMinMax(const std::vector<int> &vec) {
  int min = INT_MAX, max = INT_MIN;
  for (auto x : vec) {
    // ...
  }
  return ???
}
```

返回多个值

C 里讲过的办法：创建一个结构体

- 在 C++ 里就是一种简单的类。

```
struct MinAndMax {
  int min;
  int max;
};
MinAndMax getMinMax(const std::vector<int> &vec) {
  int min = INT_MAX, max = INT_MIN;
  for (auto x : vec) {
    // ...
  }
  return /* 这里怎么写？ */
}
```

聚合类 (aggregate)

聚合类是那些所有 non-`static` 数据成员都是 public 、没有自定义构造函数且满足一些其它条件的类。

- 核心特点一：数据成员都是 public ，看起来就如同是几个变量打包在一起，这些变量本身就是接口。
- 核心特点二：没有自定义构造函数，初始化 = 对所有成员进行初始化。
- 其它条件，例如基类都是 public 继承，没有虚函数等。

这样的类从使用的角度讲，与 C 的 `struct` 非常接近。

使用大括号列表

```
struct MinAndMax {
  int min;
  int max;
};
MinAndMax result{a, b};
```

```
MinAndMax getMinMax(/* ... */) {
  int min = INT_MAX, max = INT_MIN;
  // ...
  return {min, max};
}
```

一般地，如果用 `{e1, e2, ...}` 初始化一个 `T` 类型的对象，例如 `T x{a, b, c};`：

- 如果 `T` 是一个聚合体，那么这是一种**聚合初始化** (aggregate initialization)：`e1`，`e2`，... 依次用来初始化第一个子对象、第二个子对象、……
- 否则，如果 `T` 含有接受 `std::initializer_list<V>` 的构造函数，则这个列表被传给这个构造函数。
- 否则，就如同 `T x(a, b, c);` ，走正常的构造函数（**直接初始化** (direct initialization)）。

使用大括号列表

如果 `MinAndMax` 有一个用户提供的构造函数：

```
struct MinAndMax {
  int min;
  int max;
  MinAndMax(int min_, int max_) : min{min_}, max{max_} {}
};
```

那么 `MinAndMax` 就不再是聚合类，但是我们仍然可以用 `{a, b}` 初始化一个该类型的对象，也仍然可以 `return {min, max};`

总之，下面这种做法一定是可以避免、需要避免的：

```
MinAndMax result;
result.min = a;
result.max = b;
```

结构化绑定 (Structured binding)

```
struct MinAndMax { int min, max; };
MinAndMax getMinMax(const std::vector<int> &);
```

调用者怎么获得这个返回值?

```
auto result = getMinMax(numbers);
std::cout << result.min << ' ' << result.max << '\n';
```

这当然可以，但是有更好的办法：**结构化绑定**

```
auto [minVal, maxVal] = getMinMax(numbers);
std::cout << minVal << ' ' << maxVal << '\n';
```

结构化绑定 (Structured binding)

结构化绑定的声明可以带有 `const` 、引用等，也可以放在基于范围的 `for` 语句里：

```
enum class Gender { Male, Female };
struct PersonInfo {
  std::string name;
  Gender gender;
  int birthYear;
};
void foo(const std::vector<PersonInfo> &persons) {
  for (const auto &[n, g, by] : persons) {
    // ...
  }
}
```

`std::pair`

定义在 `<utility>` 中

一个快速、随意的数据结构

```
std::pair<T1, T2> p1{a, b}; // 用圆括号也可以
auto p2 = std::make_pair(a, b); // 等价的写法
std::pair p3{a, b}; // 编译器根据 a 和 b 的类型来推导模板参数
                    // 这里用圆括号也可以，但 {} 更 modern
```

C++17 有了 CTAD，几乎不再需要 `std::make_pair` 了。

`std::pair`

返回两个值：装进 `pair` 里就行

```
std::pair<int, std::string> foo() {
  // ...
  return {42, "hello"};
}
```

访问 `pair` 中的元素：`p.first`, `p.second`

```
auto p = foo();
std::cout << foo.first << ", " << foo.second << std::endl;
```

`std::pair`

结构化绑定也可以用于 `std::pair` ：

```
auto [ival, sval] = foo();
std::cout << ival << ", " << sval << std::endl;
```

`std::pair` 有比较运算符：按照 `.first` 为第一关键字、 `.second` 为第二关键字比较。

不要滥用 `std::pair` ！

```
using Complex = std::pair<double, double>;   // bad idea!
using Student = std::pair<std::string, int>; // bad idea!
```

重载运算符 (Operator overloading)

总结：技术

运算符重载的方式是定义一个名为 `operator@` 的函数：

- 各个运算对象从左向右依次作为这个函数的参数。
- 如果是成员函数，则最左侧的运算对象绑定到 `*this`。

不能发明新的运算符。不能为内置类型定义运算符。

不能被重载：

- `?:`，`.`，`::` 等，具有特别意义的运算符

不建议被重载：（为什么？）

- `&&`，`||`，`,`，`&` (address-of)

总结：技术

运算符重载的方式是定义一个名为 `operator@` 的函数：

- 各个运算对象从左向右依次作为这个函数的参数。
- 如果是成员函数，则最左侧的运算对象绑定到 `*this`。

不能发明新的运算符。不能为内置类型定义运算符。

不能被重载：

- `?:`，`.`，`::` 等，具有特别意义的运算符
- `?:` 无法被重载，因为它有一个运算对象不被求值，这一点用函数传参不可能做到

不建议被重载：（为什么？）

- `&&`，`||`，`,`，`&` (一元取地址运算符)
- `&&` 和 `||` 的短路求值会失效。`,` 和 `&` 本来就能作用于一切类型。

总结：技术

重载的行为需要和内置的行为保持某种程度上的一致，除非你有很好的理由不这么做

- `++i` 返回 `i` 的引用，`i++` 返回递增前的 `i` 的一份拷贝。
- 赋值、复合赋值都返回左侧运算对象的引用。
- 解引用 `*p` 通常返回左值引用
- 比较运算符需要符合逻辑：`a == b` 意味着 `b == a`，`a != b` 意味着 `!(a == b)`。

C++20 的比较运算符的相关规则发生了巨大的变化：

- 允许用 `=default` 让编译器合成
- 新增了 `operator<=>` 用来定义数学上对应的**序关系**（partial ordering, strong ordering, weak ordering）
- 定义了 `operator==` 和 `operator<=>` 后编译器会合成其它关系运算符，还会照顾对称性。

总结：技术

通常是成员：`++`，`--`，`*` (解引用)，`->`，`=` (赋值，必须是成员)，各种复合赋值运算符（`+=`，`-=`，...）

成员或非成员皆可：比较运算符 `<`，`<=`，`>`，`>=`，`==`，`!=`，算术运算符等

如果需要左侧运算对象也能隐式转换，则它不能是成员。

- `r == 1` 被视为 `r.operator==(1)`，即 `r.operator==(Rational(1))`
- 但 `1 == r` 会被视为 `1.operator==(r)`（显然是错误），不会被进一步推断为 `Rational(1).operator==(r)`

不要漏 `const`！

```
class Rational {
public:
  bool operator==(const Rational &) const;
};
bool operator!=(const Rational &, const Rational &);
```

非成员有两个 `const`，成员也必然有两个 `const`，只是其中一个变成了这个成员函数的 qualifier。

特殊的运算符：后置递增 `i++`

```
class Rational {
public:
  Rational &operator++() { /* ... */ return *this; }
  Rational operator++(int) {
    // 若无特殊情况，后置递增运算符几乎总是这样写
    auto tmp = *this; // 拷贝原来的对象
    ++*this;          // 真正的“递增”由前置版本完成
    return tmp;       // 返回递增前的拷贝
  }
};
```

- `int` 参数：**仅仅是为了区分前置版本和后置版本**，没有实际意义，也不需要用到，自然也就没有名字。
  - 如果一个参数没有被用到，它的名字可以不被写出来，也可以标上 `[[maybe_unused]]`
- `++i` 被视为 `i.operator++()`，`i++` 被视为 `i.operator++(0)`。

特殊的运算符：`->`

```
class SharedPtr {
public:
  Object &operator*() const;
  Object *operator->() const {
    // 若无特殊情况，箭头运算服几乎总是这样写，以使 p->mem 和 (*p).mem 等价
    return std::addressof(this->operator*());
  }
};
```

- 为了让 `p->mem` 和 `(*p).mem` 等价，`operator->` 几乎总是应该这样定义。
  - 这个运算符有一点怪（比如，它是一元的还是二元的？）。不用太纠结，实在好奇可以上网查。
- 注意在本例中 `operator*` 和 `operator->` 都是 `const`：它们都允许在 `const SharedPtr` 上调用。

避免重复：比较运算符

定义 `operator==` 和 `operator<`，让剩下四个依赖于它们。

```
bool operator!=(const Rational &lhs, const Rational &rhs) {
  return !(lhs == rhs);
}
bool operator>(const Rational &lhs, const Rational &rhs) {
  return rhs < lhs;
}
bool operator<=(const Rational &lhs, const Rational &rhs) {
  return !(lhs > rhs); // 你只会 lhs < rhs || lhs == rhs ?
}
bool operator>=(const Rational &lhs, const Rational &rhs) {
  return !(lhs < rhs);
}
```

避免重复：算术运算符

定义一元负号 `-` 和 `+=`，那么 `+`，`-`，`-=` 都可以用它们实现。

```
class Rational {
public:
  Rational &operator-=(const Rational &rhs) {
    return *this += -rhs;
  }
};
Rational operator+(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) += rhs;
}
Rational operator-(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) -= rhs;
}
```

输入、输出运算符：`operator<<` 和 `operator>>`

首先搞清楚：输入流（例如 `std::cin`）的类型是 `std::istream`，输出流（例如 `std::cout`）的类型是 `std::ostream`。

这两个运算符应该是成员还是非成员？

输入、输出运算符

`operator<<` 和 `operator>>`

首先搞清楚：输入流（例如 `std::cin`）的类型是 `std::istream`，输出流（例如 `std::cout`）的类型是 `std::ostream`。

这两个运算符只能是非成员：你无法给 `std::istream` 和 `std:ostream` 添加成员。

- 若一个运算符是成员，它只可能是最左侧运算对象的成员。

`std::istream` 和 `std::ostream` 都不能拷贝，必须按引用传递，而且不能是常量引用。

```
std::istream &operator>>(std::istream &, Rational &r);
std::ostream &operator<<(std::ostream &, const Rational &r);
```

- 为了支持连续的输入输出 `std::cin >> a >> b >> c`，这两个运算符需要把左侧运算对象返回出来。

输入、输出运算符

输入运算符需要考虑输入错误的情况。

```
struct Vec3 {
  double x_, y_, z_;
  double l2_norm_;
};
std::istream &operator>>(std::istream &is, Vec3 &v) {
  is >> v.x_ >> v.y_ >> v.z_;
  if (!is) // 如果输入发生错误，要将对象置于有效的状态。
    v.x_ = v.y_ = v.z_ = 0;
  v.l2_norm_ = std::sqrt(v.x_ * v.x_ + v.y_ * v.y_ + v.z_ * v.z_);
  return is;
}
```

输入、输出运算符

不能直接将 `is`、`os` 替换为 `std::cin` 和 `std::cout`：除它们之外还有其它的流对象

```
std::ifstream file("infile.txt");
int x, y, z;
file >> x >> y >> z;
```

由于 `std::ifstream` 继承自 `std::istream`，它可以被 `std::istream &` 绑定

避免重载运算符的滥用

函数的名称可以透露很多信息，而运算符不行：

- `a * b` 和 `dot_product(a, b)`
- `a < b` 和 `compare_by_id(a, b)`
- `a + b` 和 `concat(a, b)`

避免滥用重载运算符：除非这个运算符有唯一明确合理的定义，否则不要定义它。

- 为什么 `std::string` 可以用 `+` 连接，而 `std::vector` 不行？
- MATLAB 的 `a * b` 和 `a .* b` 的区别

`operator+` 应该加而不是减，`operator<` 应该是“小于”而非“大于”。

函数调用运算符 `operator()`

```
struct Adder {
  int operator()(int a, int b) const {
    return a + b;
  }
};
std::cout << Adder{}(2, 3) << std::endl; // 5
```

- `Adder{}` 创建了一个 `Adder` 类型的对象。
  - 也可以写作 `Adder()` 。
- `Adder{}(2, 3)` 相当于 `Adder{}.operator()(2, 3)`

一般地，`f(arg1, arg2, ...)` 被视为 `f.operator()(arg1, arg2, ...)` 。

* **分清这里的各个括号的含义！** 要理解每个括号所发挥的功能，不要瞎猜。

类型转换运算符

除了接受单个参数的构造函数，我们还有另一种方式自定义类型转换：

```
struct Rational {
  int n, d;
  operator double() const { // 定义了从 Rational 到 double 的类型转换
    return static_cast<double>(n) / d;
  }
};

Rational r{3, 2};
double d = r; // 1.5
```

类型转换运算符

```
struct Rational {
  int n, d;
  operator double() const {
    return static_cast<double>(n) / d;
  }
};
Rational r{3, 2};
double d = r; // 1.5
```

- 函数名是 `operator Type`
- 不接受参数，不写返回值类型（因为返回值类型就是 `Type`）
- 通常是 `const`：类型转换一般不应改变这个对象本身。

类型转换运算符

`std::istream` 对象可以放在条件部分，来判断这个输入流正不正常：

```
std::cin >> a >> b;
if (!std::cin)
  handle_input_failure();
```

所以标准库定义了 `std::istream` 向 `bool` 的类型转换，对吗？

```
class istream {
  bool fail, bad;
public:
  operator bool() const {
    return !fail && !bad;
  }
};
```

类型转换运算符

标准库定义了 `std::istream` 向 `bool` 的类型转换，对吗？

```
class istream {
  bool fail, bad;
public:
  operator bool() const { return !fail && !bad; }
};
istream cin;
```

下面的代码居然也乐呵呵地编译了！为什么？

```
int ival;
cin << ival; // 写反了！
```

类型转换运算符

标准库定义了 `std::istream` 向 `bool` 的类型转换，对吗？

```
class istream {
  bool fail, bad;
public:
  operator bool() const { return !fail && !bad; }
};
istream cin;
```

下面的代码居然也乐呵呵地编译了！为什么？

```
int ival;
cin << ival; // cin 隐式转换成 bool 类型，又提升成 int
             // 这个 << 其实是作用于两个 int 的移位运算符！
```

类型转换运算符

为了避免 `std::cin << ival` 通过编译，在 C++11 以前标准库定义的是向 `void *` 的类型转换：在输入流正常的时候返回一个非空指针，不正常的时候返回空指针。

(since C++11): `explicit` 类型转换运算符

```
struct Rational {
  int n, d;
  explicit operator double() const {
    return static_cast<double>(n) / d;
  }
};
Rational r{3, 2};
double d = r; // 错误：隐式转换
double d2 = static_cast<double>(r); // 正确
```

类型转换运算符

为了避免 `std::cin << ival` 通过编译，在 C++11 以前标准库定义的是向 `void *` 的类型转换：在输入流正常的时候返回一个非空指针，不正常的时候返回空指针。

(since C++11): `explicit` 类型转换运算符

```
class istream {
  bool fail, bad;
public:
  explicit operator bool() const { return !fail && !bad; }
};
istream cin;
```

从 `istream` 向 `bool` 的类型转换必须显式发生。

那 `if (cin)` 还能用吗?

Contextual conversions

下列情况中，`e` 向 `bool` 的类型转换允许使用 explicit 类型转换运算符：

- `if (e)`，`while (e)`，`for (xxx; e; xxx)`
- `!e`，`e && e`，`e || e`，`e ? a : b`
- （你们没学过的）`static_assert(e)`，`noexcept(e)`，`explicit(e)` (since C++20)

因此 `if (cin)` 也就没问题了。

避免类型转换运算符的滥用

```
struct Rational {
  int n, d;
  operator double() const {
    return static_cast<double>(n) / d;
  }
  operator std::string() const {
    return std::to_string(n) + '/' + std::to_string(d);
  }
};

Rational r{3, 2};
std::cout << r << std::endl; // 1.5 还是 3/2 ?
```

避免类型转换运算符的滥用

更好的设计：定义成普通的函数

```
struct Rational {
  int n, d;
  auto to_double() const {
    return static_cast<double>(n) / d;
  }
  auto to_string() const {
    return std::to_string(n) + '/' + std::to_string(d);
  }
};
```

一般来说，只为某些特殊的行为定义类型转换，尤其是不要轻易定义向内置类型的类型转换！

类型转换引发的二义性

```
struct A {
  A(const B &);
};
struct B {
  operator A() const;
};

B b{};
A a = b; // 到底是调用了 `A::A(const B &)` 还是 `B::operator A() const`?
```

类型转换引发的二义性

当类型转换遇上重载决议，情况就彻底乱套了...

```
struct Rational {
  operator double() const;
};
struct X {
  X(const Rational &);
};
void foo(int);
void foo(X);
Rational r{3, 2};
foo(r); // 到底是先转成 double 然后转成 int 匹配 foo(int)
        // 还是直接转成 X 匹配 foo(X)?
```

避免这样的情况发生！ 不要滥用类型转换运算符。