CS100 Recitation 9

Contents

- Value Categories
- C++ Class Review

Motivation

Motivation

Question: How many objects are created by the following initializations?

```
struct Data {
    Data(size_t s) {printf("Default\n");};
    Data(const Data&) {printf("Copy\n");}
    Data(Data&&) {printf("Move\n");}

    size_t s;
    int* b;
};

const Data getData(size_t s){
    return Data(s);
}
```

```
auto d1 = Data(42);

auto d2 = std::move(d1);

auto d3 = getData(42);

auto d4 = std::move(getData(42));
```

Motivation

```
auto d1 = Data(42);
// 1 object created (constructor)

auto d2 = std::move(d1);
// 1 objects: move constructor invoked

auto d3 = getData(42);
// 1 object: copy elision

auto d4 = std::move(getData(42));
// 2 objects: const-qualification forces copy constructor
```

- Value categories describe how the compiler interprets expressions in a program.
- A value category is a property of an expression, not of an object.

Overview

- Value categories were **inherited** from C, which already had the notion of an “lvalue expression.”
- They originally referred to an expression’s position in assignment:
  `auto a = int(42);`
  - **lvalue** (left value): appears on the left side of `=`
  - **rvalue** (right value): appears on the right side of `=`
- The value category of an expression determines
  - **Lifetime** – Is it a temporary? Can it be moved from? Will it outlive the full-expression?
  - **Identity** – An object has identity if its address can be taken and used reliably.
- Value categories critically influence **performance** and **overload resolution**.

Value Category vs. Type

```cpp
struct Data {
  Data (int x);
  int x_;
};

void foo(Data&& x){
  x = 42;
}
```

```cpp
Data a = 42;

Data&  lval_ref_a = a;  // lvalue reference
Data&& rval_ref_a = 42; // rvalue reference

foo(rval_ref_a); // Error: lvalue!
foo(Data(73));   // OK
```

An object’s type is distinct from its value category:

- `rval_ref_a` **type**: “rvalue reference to `Data`”
- The **expression** `rval_ref_a` is an **lvalue**

Value Category vs Scope (Context)

```cpp
struct Data {
  Data (int x);
  int x_;
}

void foo(Data &&x) {
  x = 42;
}

int main() {
  foo(Data(73));

  return 0;
}
```

The entity can have different Value Category in different contexts.

- During a function call: (`foo(Data(73))`)
  - Step 1: Calls constructor, creates an unamed temp `Data(73)`.
  - Step 2: `Data(73)` binds to the rvalue reference `x`.
  - Step 3: The entity which used to be `Data(73)` has a name `x`, therefore, in the scope of `foo`, `x` is now an lvalue.

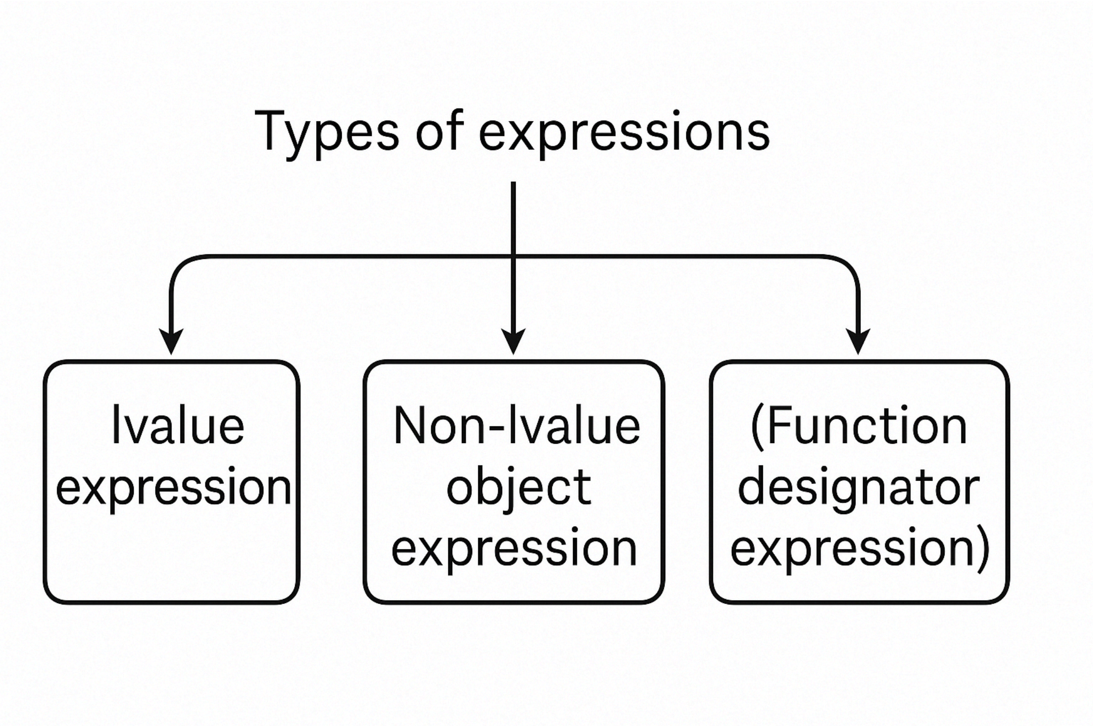
Evolution of Value Categories

Value-category rules have evolved across C++ versions, driven by references, move semantics, and copy-elision improvements.

C language

Three kinds of expression:

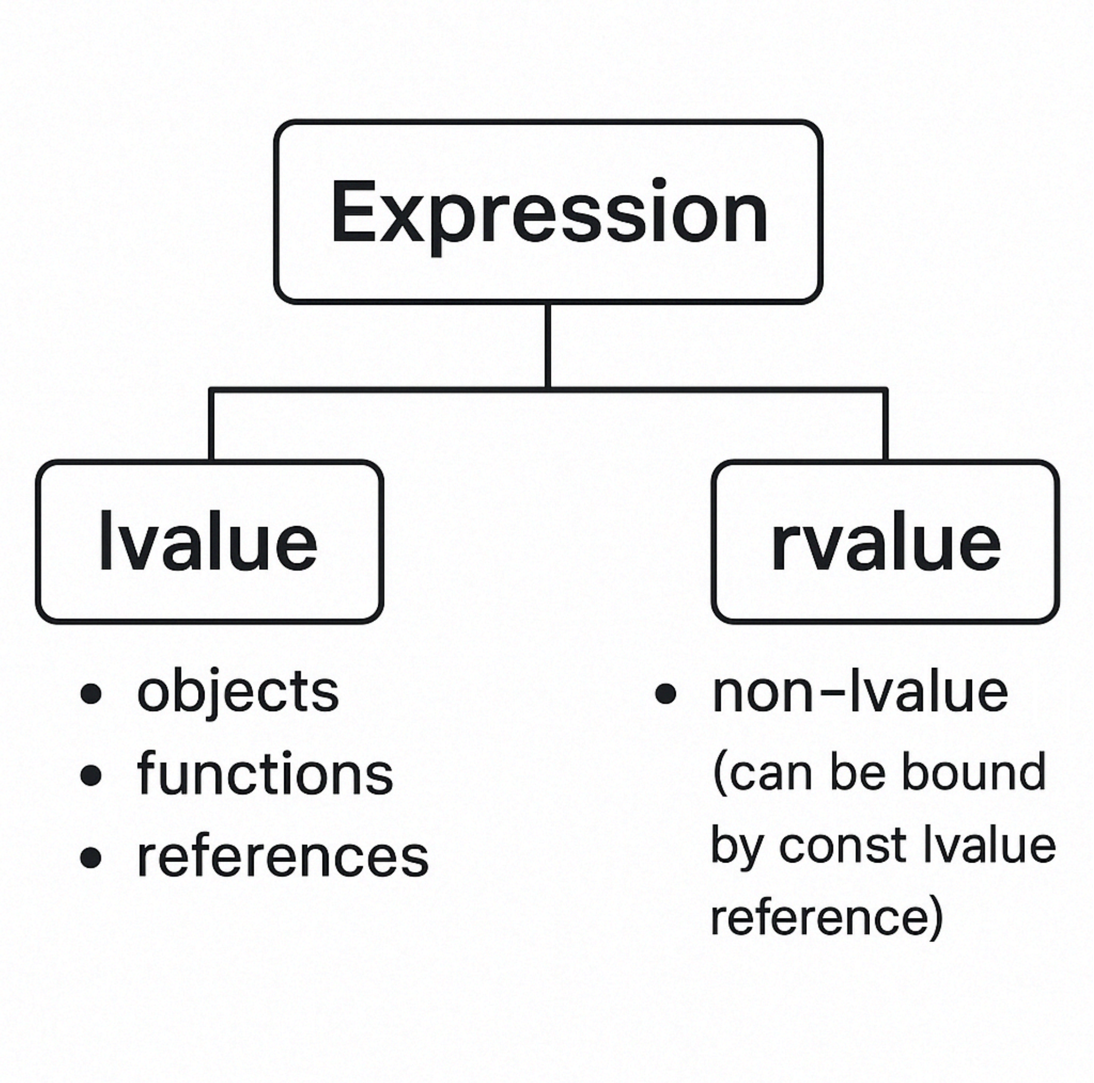
- lvalue expression
- non-lvalue object expression
- (function-designator expression)



C++98 – Added (lvalue) References

Expressions are either an **lvalue** or an **rvalue**.

- **lvalue**: objects, functions, references
- **rvalue**: non-lvalues (can bind to a `const` lvalue reference)



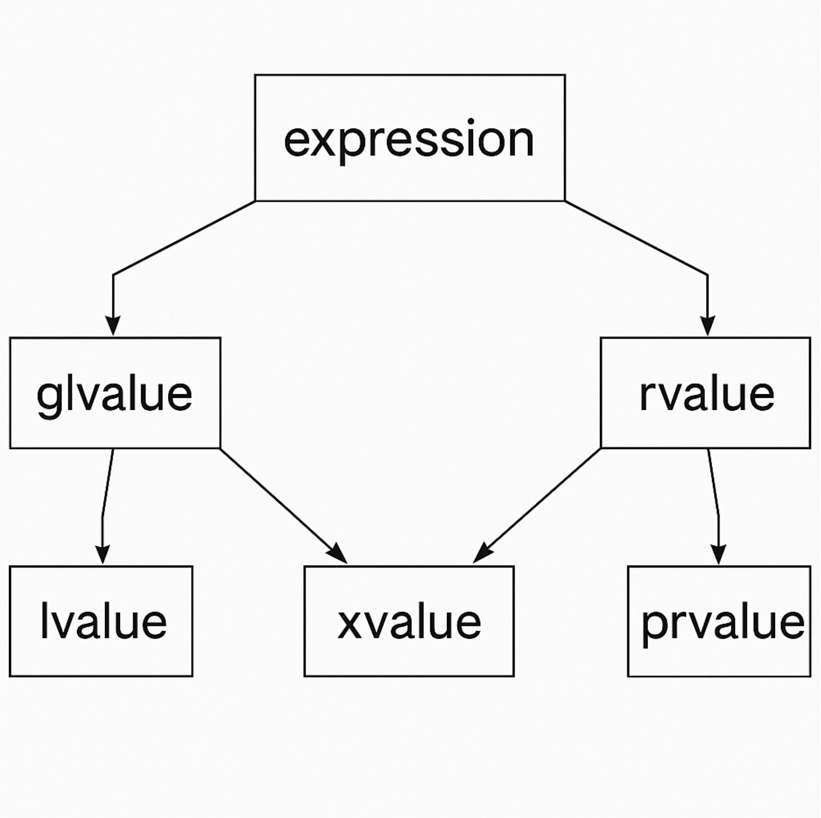
C++11 – Added Rvalue References & Move Semantics

	Has identity (glvalue)	Has no identity
Cannot be moved from	lvalue	—
Can be moved from (rvalue)	xvalue	prvalue

- **Lvalues** (locator values): designate an object (a specific memory location)
  `x`, `arr[0]`, `*ptr`
- **Prvalues** (pure rvalues): represent a value without identity
  `42`, `a + b`, `func()`
- **Xvalues** (expiring values): denote objects nearing the end of their lifetime
  `std::move(a)`, `static_cast<int&&>(a)`

C++11 – Refined Taxonomy

- **Main categories**
  - **glvalue** – expression whose evaluation determines the identity of an object or function
  - **rvalue** – either a prvalue or an xvalue
- **Sub-categories**
  - **lvalue** – glvalue that is not an xvalue
  - **xvalue** – glvalue referring to an object whose resources can be reused
  - **prvalue** – expression whose evaluation initialises an object or computes an operand value



C++17 – Guaranteed Copy-elision

- **P0135** mandates copy-elision in more cases, constructing the result object **in-place** rather than creating temporaries.

	Has identity (glvalue)	Has no identity
Cannot be moved from	lvalue	—
Can be moved from (rvalue)	xvalue	prvalue (after materialisation)

Rvalue & Lvalue

- **Lvalue** – denotes a persistent object (an addressable entity in memory).
- **Rvalue** – denotes only a value (often a temporary), not a distinct object.

The next two slides list typical examples.

Lvalues – Expressions That Denote Objects

- **Object access**
  - Pointer dereference: `*ptr`
  - Array subscript: `arr[i]`
- **Operators yielding lvalues**
  - Pre-increment/decrement: `++i`, `--j` (so `++i = 42;` is valid)
  - Assignment: `a = b` yields an lvalue referring to `a`
- **Named entities**
  - Variables, functions, static data
  - String literals (e.g. `"hello"`) in static memory

Rvalues – Expressions That Denote Values

- **Temporary objects** – result of a function call

```
std::string fun();
std::string s = fun();  // fun() yields a temporary rvalue
```

- **Literal constants** – integer, floating-point, character literals
- **Arithmetic/logical expressions** – `a + b`, `x * y`, `cond1 && cond2`
- **Functional-style casts** – `Type(args…)` constructs a temporary

```
std::string temp = std::string(10, 'c');
int n = int(x);
```

> **Note:** Built-in casts and arithmetic always produce rvalues.

Questions

Quiz, Spring 2023

Let `ival` be an `int` and `ptr` a pointer. Select the expressions that yield an **rvalue**.

A. `++ival` B. `ival++` C. `*&ptr` D. `&*ptr` E. `ptr[ival]` F. `*(ptr + ival)`

Solution

`++ival` returns a reference to `ival` → lvalue.

`ival++` returns the original value → **rvalue**.

`*&ptr` returns a reference to `ptr` → lvalue.

`&*ptr` yields the address stored in `ptr`; the result is a value, not `ptr` itself → **rvalue**.

`ptr[ival]` and `*(ptr + ival)` are equivalent; both yield references to `ptr[ival]` → lvalues.

Binding to References

Expressions with different value categories bind to different reference types.

	Binds lvalues	Binds rvalues
lvalue reference	yes	no
`const` lvalue reference	yes	yes
rvalue reference	no	yes
`const` rvalue reference	no	yes

Binding extends object lifetime:

- `const` lvalue reference – extends lifetime (no modification allowed)
- rvalue reference – extends lifetime of a temporary

Tools for Handling Value Categories

- `std::move`
- `std::forward`
- …

`std::move`

Utility function that produces an **xvalue** of type `T&&`.

- Equivalent to `static_cast` to a `T` rvalue reference type

```cpp
// Overloads
void foo(int& x) {
  std::cout << "int&";
}

void foo(const int& x) {
  std::cout << "const int&";
}

void foo(int&& x) {
  std::cout << "int&&";
}
```

```cpp
int a = 73;
int& b = a;
const int& c = a;
const int&& d = 42;
foo(std::move(b));
// std::move(b) is an int&&
// → calls foo(int&&)
foo(std::move(c));
// std::move(c) is a const int&&
// → calls foo(const int&)
foo(std::move(d));
// std::move(d) is a const int&&
// → calls foo(const int&)
```

C++ Class Review

- Dynarray and Class Review
- Operator Overloading
- A Class Example

Dynarray and Class Review

- Members
  - Data members
  - Member functions
- Constructors
  - Default constructor
  - *Custom* general constructors
  - Copy constructor [copy-control]
  - Move constructor [copy-control]
- Destructor [copy-control]
- Operator Overloads
  - Copy-assignment operator [copy-control]
  - Move-assignment operator [copy-control]

Members

- Data members
- Member functions

Data Members

```cpp
class Dynarray {
private:
  int*        m_storage;
  std::size_t m_length;
};
```

- `private` is an **access specifier**. It governs access to all subsequent members until the next specifier or the end of the class.
- **Why hide data members?**
  Encapsulation protects an object's integrity and reduces coupling by hiding implementation details behind a controlled interface, making code more robust and maintainable.
  See "Effective C++" Item 22 (mandatory reading).

Data Members – Class Invariants

```
class Dynarray {
private:
  int*        m_storage;
  std::size_t m_length;
};
```

- There is an implicit invariant: `m_length` equals the size of the memory allocated for `m_storage`.
- Allowing users to modify these variables arbitrarily could violate this property, creating invalid objects whose member functions assume the invariant holds. Hence construction, modification, and assignment must be funnelled through ***controlled interfaces*** (constructors, setters/getters, copy/move assignment). Destruction must mirror construction – the essence of the **Rule of Three/Five**.

Member Functions: `this`

- The `.` operator accesses members (especially functions, since data members are often `private`) through an object.
- The `->` operator accesses members through a **pointer** to an object.

```
class Person {
private:
  std::string name;

public:
  Person(const std::string& name) : name(name) {}
  void greet() {
    std::cout << "Hello, I'm " << name << std::endl;
  }
};
```

Member Functions: `this`

Why do two distinct objects produce different output when the same member function is called?

```cpp
int main() {
  Person alice("Alice");
  Person bob("Bob");
  alice.greet(); // Hello, I'm Alice
  bob.greet();   // Hello, I'm Bob

  return 0;
}
```

- The compiler must know **which** object invoked the function so it can access that object's data.
- It achieves this via the implicit `this` **pointer**.

Member Functions: `this`

`this` is an implicit pointer passed to every non- `static` member function, holding the address of the invoking object.

- Inside the function you can write `this->member`, but the `this->` can usually be omitted.
- A **low-level `const` qualifier** can be added to `this` by marking the function `const`.

```
void greet() const {
  std::cout << "Hello, I'm " << name << std::endl;
}
```

Member Functions: `this` & `const`

- If `ptr` has type `const T*` and `T` contains a data member `m` of type `U`, then `ptr->m` has type `const U`. **Const-ness is transitive.**
- A `const` object can invoke only `const` member functions; non-`const` functions cannot be called.

> **Best practice:** If a member function should not modify the object, declare it `const` to let the compiler enforce correctness.

Dynarray: Constructors

```cpp
class Dynarray {
  int* m_storage;
  std::size_t m_length;
public:
  Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}
};
```

- Constructors have **no return type**. The body may contain `return;` but must not return a value.
- All data members are initialised **before** entering the constructor body.
- Members are initialised in the **order of declaration**, not in the order shown in the *initializer list*.

Dynarray: Constructors

```cpp
Dynarray() noexcept : m_storage(nullptr), m_length(0) {}

explicit Dynarray(std::size_t n)
  : m_storage(new int[n]{}), m_length(n) {}

Dynarray(std::initializer_list<int> init)
  : Dynarray(init.begin(), init.end()) {}

Dynarray(const std::vector<int>& vec)
  : Dynarray(vec.data(), vec.size()) {}
```

- Classes are typically given **multiple constructors** to allow flexible initialisation.
- Always use an initializer list. Not all types are default-constructible or assignable.
  - Counterexample: `T &`, `const`

Dynarray: Destructors

```
class Dynarray{
private:
  std::size_t m_length;
  int* m_storage;
public:
  Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}
  ~Dynarray() {
    delete[] m_storage;
  }
};
```

- A destructor is automatically invoked when an object goes out of scope or is deleted.
- It releases resources acquired during the object’s lifetime.

Dynarray: Destructors

If no destructor is declared, the compiler synthesises one:

- It is `public` by default.
- It has an empty body `{}` and simply destroys data members.

Member Functions and Usage

```cpp
// …
std::size_t size() const { return m_length; }
bool empty() const { return m_length == 0; }

int& at(std::size_t i) { return m_storage[i]; }
const int& at(std::size_t i) const { return m_storage[i]; }
};

void print(const Dynarray& a) {
  for(std::size_t i = 0; i != a.size(); ++i)
    std::cout << a.at(i) << " ";
  std::cout << std::endl;
}

void reverse(Dynarray& a) {
  for(std::size_t i = 0, j = a.size() - 1; i < j; ++i, --j)
    std::swap(a.at(i), a.at(j));
}
```

Example Usage

```cpp
void print(const Dynarray& a) {
  for (std::size_t i = 0; i < a.size(); ++i)
    std::cout << a.at(i) << " ";
  std::cout << std::endl;
}

void reverse(Dynarray& a) {
  for (std::size_t i = 0, j = a.size() - 1; i < j; ++i, --j)
    std::swap(a.at(i), a.at(j));
}
```

Dynarray: Copy Constructor

```
Dynarray(const Dynarray& other)
  : m_storage(new int[other.m_length]),
    m_length(other.m_length) {
  for (std::size_t i = 0; i < m_length; ++i)
    m_storage[i] = other.m_storage[i];
}
```

Dynarray: Copy-Assignment Operator

`a = b` is equivalent to `a.operator=(b)`

```
Dynarray& operator=(const Dynarray& other) {
  if (this != &other) {
    int* new_data = new int[other.m_length];
    for (std::size_t i = 0; i < other.m_length; ++i)
      new_data[i] = other.m_storage[i];
    delete[] m_storage;
    m_storage = new_data;
    m_length  = other.m_length;
  }
  return *this;
}
```

Dynarray: Copy-Assignment Variants

- **Synthesised** copy-assignment

```
Dynarray& operator=(const Dynarray& other){
    m_storage = other.m_storage;
    m_length  = other.m_length;
    return *this;
}
```

- **Defaulted**

```
Dynarray& operator=(const Dynarray& other) = default;
```

- **Deleted**

```
Dynarray& operator=(const Dynarray& other) = delete;
```

Dynarray: Move Constructor

```
Dynarray(Dynarray&& other) noexcept
  : m_storage(other.m_storage),
    m_length(other.m_length) {
  other.m_storage = nullptr;
  other.m_length  = 0;
}
```

`noexcept` promises that the function does not throw; this enables the STL to prefer non-throwing moves over potentially throwing copies.

Dynarray: Move-Assignment Operator

```
Dynarray& operator=(Dynarray&& other) noexcept {
  if (this != &other) {
    delete[] m_storage;
    m_storage        = other.m_storage;
    m_length         = other.m_length;
    other.m_storage = nullptr;
    other.m_length  = 0;
  }
  return *this;
}
```

Subscript Operator

```
int& operator[](std::size_t n) { return m_storage[n]; }
const int& operator[](std::size_t n) const { return m_storage[n]; }
```

- Provides intuitive element access akin to built-in arrays.

Operator Overloading – Programming Technique

Operator Overloading

The way to overload an operator is by defining a function named `operator@`:

- Operands are passed left-to-right.
- If the function is a **member**, the leftmost operand binds to `*this`.

You cannot create new operators or overload operators for built-in types.

Operator Overloading

Operators that **cannot** be overloaded:

- `?:`, `.`, `::`, etc.
  `?:` cannot be overloaded because one operand is unevaluated, which cannot be emulated.

Operators **not recommended** for overloading:

- `&&`, `||`, `,`, unary `&`
  Overloading `&&` / `||` destroys short-circuit semantics; `,` and `&` already work for all types.

Operator Overloading – Consistency with Built-in Behaviour

Overloaded operators should behave like their built-in counterparts unless there is a compelling reason otherwise.

- `++i` returns a reference; `i++` returns the old value.
- Assignment returns a reference to the left operand.
- Dereference `*p` usually yields an lvalue.
- Comparison operators must satisfy logical properties (`==` is symmetric, `!=` is its negation).

Operator Overloading – Member vs. Non-member

- Commonly implemented as **members**:
  `++`, `--`, unary `*` (dereference), `->`, assignment `=`, and compound assignments (`+=`, `-=`, …).
- Often implemented as **non-members**:
  Comparison operators `<`, `<=`, `>`, `>=`, `==`, `!=`, arithmetic operators, etc.

> **If the left operand requires implicit conversion, the operator cannot be a member.**
>
> - `r == 1` ⇒ `r.operator==(1)` ⇒ `r.operator==(Rational(1))`
> - `1 == r` ⇒ parsed as `1.operator==(r)` (invalid), and the compiler will not consider converting `1` into `Rational(1)` for this call.

Don't Forget `const`

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

- Non-member operators take two **const references**.
- Member operators take one const parameter and are themselves `const`.

Special Operator: Post-increment `i++`

```cpp
class Rational {
public:
  Rational& operator++() { /* … */ return *this; }      // prefix
  Rational  operator++(int) {                           // postfix
    auto tmp = *this;
    ++*this;
    return tmp;
  }
};
```

- The dummy `int` parameter distinguishes postfix from prefix.

Special Operator: `->`

```
class SharedPtr {
public:
  Object& operator*()  const;
  Object* operator->() const {
    // Ensures p->mem ≡ (*p).mem
    return std::addressof(this->operator*());
  }
};
```

`operator->` is almost always defined in terms of `operator*`, and both are `const`.

Avoid Duplication: Comparison Operators

Implement `==` and `<`, then derive the rest:

```
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); }
bool operator>=(const Rational& lhs, const Rational& rhs) { return !(lhs < rhs); }
```

Avoid Duplication: Arithmetic Operators

Define unary `-` and `+=`; derive the rest:

```
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;
}
```

Input/Output Operators

I/O operators **must** be non-member functions because you cannot modify `std::istream` or `std::ostream`:

- Input streams (e.g., `std::cin`) are of type `std::istream`.
- Output streams (e.g., `std::cout`) are of type `std::ostream`.
- Both are non-copyable and must be passed by reference (not `const`).

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

To support chaining (`std::cin >> a >> b >> c`), each operator returns the stream.

I/O Operators – Error Handling

```
struct Vec3 {
  double x_, y_, z_, l2_norm_;
};
std::istream& operator>>(std::istream& is, Vec3& v) {
  is >> v.x_ >> v.y_ >> v.z_;
  if (!is) {          // on input failure, restore a valid state
    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;
}
```

I/O Operators – Error Handling

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

- `std::ifstream` inherits from `std::istream`, so it binds to `std::istream&`.
- Do not assume only `std::cin` and `std::cout` are used.

Avoid Misuse of Operator Overloading

Function names convey intent; operators do not:

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

Only overload operators when their meaning is clear and unambiguous:

- Why can `std::string` use `+` but `std::vector` cannot?
- MATLAB distinguishes matrix multiplication `a * b` from element-wise `a .* b`.

Ensure `operator+` adds, and `operator<` means "less than."

Function Call Operator `operator()`

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

- `Adder{}` (or `Adder()`) creates an `Adder` object.
- `Adder{}(2, 3)` is equivalent to `Adder{}.operator()(2, 3)`.
- **Distinguish each set of parentheses and its role.**

Conversion Operators

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

Conversion Operators – Details

- The function name is `operator Type`.
- It takes no parameters; the return type is `Type`.
- Usually `const`; conversions should not modify the object.

Conversion Operators – Stream State

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

Surprisingly, this allows `cin << ival;` to compile—why?

Conversion Operators – `explicit`

Since C++11, conversion operators can be marked `explicit`:

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

- `if (cin)` still works due to contextual conversion.

Contextual Conversions

Contexts that allow `explicit` conversion to `bool`:

- `if (e)`, `while (e)`, `for (...; e; ...)`
- `!e`, `e && e`, `e || e`, `e ? a : b`
- (Since C++20) `static_assert(e)`, `noexcept(e)`, `explicit(e)`

Avoid Misuse of Conversion Operators

```
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);
  }
};
std::cout << r << std::endl; // 1.5 or "3/2"?
```

Avoid Misuse of Conversion Operators – Better Design

```cpp
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);
  }
};
```

Reserve conversion operators for truly unambiguous cases.

Ambiguity from Conversion Operators

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

B b{};
A a = b; // Calls A::A(const B&) or B::operator A()?
```

Ambiguity from Conversion Operators

```cpp
struct Rational {
  operator double() const;
};
struct X {
  X(const Rational&);
};
void foo(int);
void foo(X);
Rational r{3, 2};
foo(r); // double→int for foo(int), or Rational→X for foo(X)?
```

Avoid such ambiguity!

An Example C++ Class

CS100 Recitation 9