Part 4 — Type Casts, Static, Operator Overloading

Type Casts — Static Cast

- Casting operations disable the compiler type check
- Major cause for porting issues! Minimize their usage!
- C++ uses long keywords to discourage using them and to let you find them easily (e.g. with grep)

`static_cast`

- Used for standard (numerical) conversions

```cpp
int i;
double d;
i = static_cast<int>(d);        // same as  i = (int)d; (rounds towards 0)
```

Compile-time operator — no run-time check

Do not use for pointer up/down-casts (see `dynamic_cast` below)

Type Casts — Reinterpret Cast

- Conversion from one pointer type to any other pointer type
- **Dangerous**! **Avoid**! Can result in unportable code
- But sometimes useful, to get byte-level access
- Can also convert pointers to ints and vice versa DONT!

Example 1:

```
int a[100];
// char *p = a; doesn't work: type error
char *p = reinterpret_cast<char*>(a);    // get direct access to 400 bytes
```

Example 2: save integer to file in portable fashion (independent of the machine architecture) — assuming `sizeof(X)` is identical.

```
int x;
char *p = reinterpret_cast<char*>(&x);
// write low byte first
if (little_endian_machine) {
  for (int i=0; i < sizeof(x); ++i) write_byte(p[i]);
} else {
  for (int i=sizeof(X)-1; i >= 0; --i) write_byte(p[i]);
}
```

Type Casts — Const Cast

- Toggles status: const $\leftrightarrow$ non-const
- Used for changing nonessential data members in const functions (which
- can better be accomplished by adding mutable keyword)

Suppose we want to keep track of how often a const function is called:

```
class X {
public:
    int get_index() const {
        // increment counter,
        // preserve constness
        // ++get_n; doesn't work because get_index is const

        ++const_cast<X*>(this)->getn;
        ...
    }
private:
    int get_n;      // how often was get_index called?
};
```

Why does this work?

Type Casts — Const Cast

- In const functions `this` has type `const X*`
- So, `++this->get_n` is not allowed
- "`const_cast<X*>(this)`" has type X* (const stripped away)
- Therefore,

```
++const_cast<X*>(this)->get_n
```

works

Weird concept … better solution:

```cpp
class X {
public:
    int get_index() const {
        ++get_n;    // allowed because get_n is mutable
        ...
    }
private:
    mutable int get_n;
};
```

Because mutable variables can be changed in const functions, they must not be essential for the object state!

Type Casts — Dynamic Cast

- Used for "*walking up and down the type hierarchy*"

```cpp
struct X {
    virtual void foo();     // makes X polymorphic
};

class Y : public X { };
...

X *px = new X;
Y *py = new Y;

// failed down-cast
// effect: pxy = nullptr, because Xs are no Ys
Y *pxy = dynamic_cast<Y*>(px);

// successful up-cast
// effect: pxy != nullptr pointing to X component because Ys are Xs
// At this point, the cast never fails because py points to a Y
X *pyx = dynamic_cast<X*>(py);
```

Because mutable variables can be changed in const functions, they must not be essential for the object state!

Type Casts — Dynamic Cast

- Useful for down-casting pointers (trying to treat them as derived class pointers)
- Beginners often (mis-)use down-casts for implementing type switches like this:

```
Shape *pShape = ...;
Circle *p = dynamic_cast<Circle*>(pShape);
if (p) {
    // pShape points to a Circle, call non-virtual Circle function
    p->draw();
}

Rectangle *q = dynamic_cast<Rectangle*>(pShape);
if (q) {
    // pShape points to a Rectangle, call non-virtual Rectangle function
    q->draw();
}
...
```

Warning: The presence of dynamic_cast usually indicates a broken class design. Use virtual functions instead!

Type Casts — Dynamic Cast

- `dynamic_cast` is a non-trivial run-time check, which may slow down your program.
- For it to work, the source type must be polymorphic.
- Internally, `dynamic_cast<T*>(p)` invokes a type graph traversal (following VFTPs) to check whether T is a base-class of the type p points to
- Returns `nullptr` if cast is illegal, and pointer to object if valid

Static Data and Functions

Using `static` outside class definitions:

```cpp
// In Foo.cpp:

static void foo() { }
// this defines a helper function local to Foo.cpp
// which is not accessible in other .cpp files

void bar() {
    static X x;
    // Object x is a persistent *global* variable which is
    // constructed when bar is executed for the first time.
    // Such static construction is thread-safe (which means
    // that multiple execution threads can call function bar
    // simultaneously) and static objects are destroyed in
    // reverse order of their construction.
}
```

Static Data and Functions

Because the initialization order of global objects isn't well defined in C++, it is a good idea to wrap global objects in access functions like so, to choose the time of construction (the first use of the function):

```
struct X {
    X() { a = 0; }
    int a;
};

X &get_x() {
    static X x;      // x constructed during first call
    return x;        // reused thereafter
}

int main() {
    get_x().a = 0;     // create static object, set a
    std::cout << get_x().a << std::endl;     // use object
}
```

Static in Class Contexts

Sometimes it is useful if all objects of a class have access to the same variable, e.g. a "*global*" class option or a counter that keeps track of how many objects have been created.

This can also save space. E.g. a shared pointer to an error-handling routine

Advantages:

- Information hiding can be enforced. Static members can be private — global variables cannot
- Static members are not entered in global namespace, limiting accidental name conflicts

Syntax: `static` qualifier in front of variable or function declaration

Static in Class Contexts — Example

```cpp
class X {
public:
    X()  { ++count; ... }
    ~X() { --count; ... }
    // static member function
    static int get_count() { return count; }
private:
    // number of X objects, shared by all X objects
    static int count;
};

int X::count = 0;      // must be defined in file X.cpp!

int main() {
    X a, b;
    cout << X::get_count() << endl;
    // output 2; note that we don't need an object
    // to call get_count --- it's a global function in class X
    return 0;
}
```

In C++17, you can define a static data member inside the class definition using `inline`, and thus does not need an out-of-class definition.

```cpp
class X {
    ...
    inline static int count = 0;
};

// X.cpp
```

```
// int X::count = 0; No longer needed
```

Operating Overloading

Goal: No difference between built-in types and class types — we want to be able to define what operators do when applied to our own classes.

We would like to write:

```
Matrix a(N,N), b(N,N), c(N,N), d(N,N);
Vector v(M);

a = b + c * d;

std::cout << a;
std::cin >> b;

v[0] = 0;            // v looks like an array - nifty!
```

C++ allows users to overload/redefine global operators such as << and class operators such as [ ]

Limits: arity (how many parameters), associativity (left or right first?), and operator precedence (3+3*4 : * evaluated first) are fixed!

Operating Precedence

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>1</td><td>::</td><td>Scope resolution</td><td rowspan="6">Left-to-right →</td></tr>
<tr><td rowspan="5">2</td><td>a++ a--</td><td>Suffix/postfix increment and decrement</td></tr>
<tr><td>*type*() *type*{}</td><td>Functional cast</td></tr>
<tr><td>a()</td><td>Function call</td></tr>
<tr><td>a[]</td><td>Subscript</td></tr>
<tr><td>. -></td><td>Member access</td></tr>
<tr><td rowspan="10">3</td><td>++a --a</td><td>Prefix increment and decrement</td><td rowspan="10">Right-to-left ←</td></tr>
<tr><td>+a -a</td><td>Unary plus and minus</td></tr>
<tr><td>! ~</td><td>Logical NOT and bitwise NOT</td></tr>
<tr><td>(*type*)</td><td>C-style cast</td></tr>
<tr><td>*a</td><td>Indirection (dereference)</td></tr>
<tr><td>&a</td><td>Address-of</td></tr>
<tr><td>sizeof</td><td>Size-of[note 1]</td></tr>
<tr><td>co_await</td><td>await-expression (C++20)</td></tr>
<tr><td>new new[]</td><td>Dynamic memory allocation</td></tr>
<tr><td>delete delete[]</td><td>Dynamic memory deallocation</td></tr>
<tr><td>4</td><td>.* ->*</td><td>Pointer-to-member</td><td rowspan="12">Left-to-right →</td></tr>
<tr><td>5</td><td>a*b a/b a%b</td><td>Multiplication, division, and remainder</td></tr>
<tr><td>6</td><td>a+b a-b</td><td>Addition and subtraction</td></tr>
<tr><td>7</td><td><< >></td><td>Bitwise left shift and right shift</td></tr>
<tr><td>8</td><td><=></td><td>Three-way comparison operator (since C++20)</td></tr>
<tr><td>9</td><td>< <= > >=</td><td>For relational operators < and ≤ and > and ≥ respectively</td></tr>
<tr><td>10</td><td>== !=</td><td>For equality operators = and ≠ respectively</td></tr>
<tr><td>11</td><td>a&b</td><td>Bitwise AND</td></tr>
<tr><td>12</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>13</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>14</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>15</td><td>||</td><td>Logical OR</td></tr>
<tr><td rowspan="8">16</td><td>a?b:c</td><td>Ternary conditional[note 2]</td><td rowspan="8">Right-to-left ←</td></tr>
<tr><td>throw</td><td>throw operator</td></tr>
<tr><td>co_yield</td><td>yield-expression (C++20)</td></tr>
<tr><td>=</td><td>Direct assignment (provided by default for C++ classes)</td></tr>
<tr><td>+= -=</td><td>Compound assignment by sum and difference</td></tr>
<tr><td>*= /= %=</td><td>Compound assignment by product, quotient, and remainder</td></tr>
<tr><td><<= >>=</td><td>Compound assignment by bitwise left shift and right shift</td></tr>
<tr><td>&= ^= |=</td><td>Compound assignment by bitwise AND, XOR, and OR</td></tr>
<tr><td>17</td><td>,</td><td>Comma</td><td>Left-to-right →</td></tr>
</table>

Operating Precedence — Example

- "`N::x.m`" means "`(N::x).m` rather than "`N::(x.m)`"
- "`*p++`" means "`*(p++)`" rather than "`(*p)++`"
- "`a + b * c`" means "`a + (b * c)`"
- "`a = b = c`" means "`a = (b = c)`"
- "`a + b + c`" means "`(a + b) + c`"
- "`i & 3 == 0`" means "`i & (3 == 0)`" Careful!
- "`a || b && c`" means "`a || (b && c)`" Careful!
- "`++++i`" means "`++(++i)`"

If in doubt, you can always force the evaluation by inserting balanced pairs of parentheses, like so:

```
(a + b) * c   or   (*p)++
```

Complex Number Example

```cpp
// defines class that represents complex numbers
// (essentially points in 2d defining the number
// field complex analysis is concerned with)
#include "Complex.h"

int main() {
    Complex a(1.0);
    Complex b(0.0, 1.0);
    Complex c;

    // arithmetic using points rather than scalars
    c = (a + b) * (a - b);
    c += Complex(4, 3);
    c = c + 3.0;                        // shorthand for + (3.0, 0.0)
    ++c;                                // means c = c + (1.0, 0)
    std::cout << c << std::endl;        // prints 8 3
};
```

Global Operators

Example C++ I/O streams

- How to define global operators such as input/output operators `<< >>` ?
- `ostream &operator<< (ostream &os, const X &rhs);`
- `istream &operator>> (istream &is, X &rhs);`

Reference to streams is returned to allow chaining such as

- `out << x << y;`
- `cin >> x >> y;`

Global Operators — Example

Attempt 1

```cpp
class Complex {      // Complex number class
    ...
private:
    float re, im;    // real and imaginary component
};

// write complex number to output stream
auto operator<<(std::ostream &os, const Complex &rhs) -> std::ostream & {
    os << rhs.re << ' ' << rhs.im;
    return os;
}

// read complex number from input stream
auto operator>>(std::istream &is, Complex &rhs) -> std::istream & {
    is >> rhs.re >> rhs.im;
    return is;
}
```

Error: re, im are private and can't be accessed outside the class

Global Operators — Example

Solution: Friends or Getters/Setters

```
using namespace std;

class Complex {
public: ...
    // gives functions access to private members
    friend ostream &operator<<(ostream &os, const Complex &rhs);
    friend istream &operator>>(istream &is, Complex &rhs);
private:
    float re, im;
};

ostream &operator<<(ostream &os, const Complex &rhs) {
    os << rhs.re << ' ' << rhs.im;
    // Alternative:
    //    os << rhs.get_re() << ' ' << rhs.get_im();
    return os;
}

istream &operator>>(istream &is, Complex &rhs) {
    is >> rhs.re >> rhs.im;
    // Alternative: float u; is >> u; rhs.set_re(u); is >> u; rhs.set_im(u);
    return is;
}

// application
Complex a;
cin >> a;  cout << a;
```

Class Operators

Class operators can be considered methods that are invoked when the lhs of a binary operation is an object and the rhs is another object or POD, or when the argument of a unary operator is an object.

The compiler internally rewrites operators into member function calls.

- `(a + b)` becomes `a.operator+(b)`
- `(a += b)` becomes `a.operator+=(b)`
- `v[i+1]` becomes `v.operator[](i+1)`
- `f(x, y)` becomes `f.operator()(x, y)`
  - `f` is called a "functor", looks like a regular function
- `++x` becomes `x.operator++()`
- `x++` becomes `x.operator++(0)`
  - `0` is a dummy variable indicating post increment

Class Operators

- `T operator++() {...}` defines the *prefix* operator
- `T operator++(int) {...}` defines the *postfix* operator
- So class operators are actually member functions, they can even be virtual!
  - `[]` supports exactly one (arbitrary) argument
  - `()` supports arbitrary number of arguments

This means that we can create objects that behave like arrays or function!

Type cast operators can also be customized:

```
class Rational {
    operator double() { return (double)num / double(den); }
};

Rational r;
cout << static_cast<double>(r) << endl;     // calls operator double()
```

Class Operators — Vector Example

```cpp
class V {
public:
    ...
    // returns reference so that elements can be changed
    int &operator[](int i) {
        check(i);
        return p[i];
    }
    // const version
    const int &operator[](int i) const {
        check(i);
        return p[i];
    }
    ...
private:
    void check(int i) const { assert(i >= 0 && i < n); }
    int *p;
    int n;
};

// in main():
V v(100);
v[3] = 0; cout << v[0];     // cool! vectors act like arrays
```

Class Operators — Complex Example

```cpp
class Complex { // Complex Number class
public:
    Complex(float r=0, float i=0) : re(r), im(i) {}
    // use default destructor; default CC+AO also work

    Complex operator+(const Complex &rhs) const;
    Complex operator+(float rhs) const;          // add float
    ...
    Complex &operator+=(const Complex &rhs);
    Complex &operator+=(float rhs);              // add float
    ...
    Complex &operator++();                       // pre++  (++c)
    Complex  operator++(int);                    // post++ (c++)
    Complex  operator-() const;                  // unary operator
    ...
    float real() const { return re; }            // gives environment
    float imag() const { return im; }            // access to data

private:
    float re, im;  // real & imaginary part
};
```

Class Operators — Complex Example

For class Complex to be fully functional, we also need global operators such as

```
Complex operator+(double lhs, const Complex &rhs);
```

to deal with asymmetries such as:

```
Complex a, b;
a = 2.0 + b;
```

which can't be handled by class operators because the lhs type is not a struct/class.

Complex Class Implementation

```cpp
#include "Complex.h"

// case:  a + b  (a,b Complex)
Complex Complex::operator+(const Complex &rhs) const {
    // computes new coordinates, copy-constructs a new
    // object and returns it to the caller
    return Complex(re + rhs.re, im + rhs.im);
}

// case: a + f  (a Complex, f float)
Complex Complex::operator+(float rhs) const {
    return Complex(re + rhs, im);
}

// executed for  a += b  (a,b Complex)
// Note: cascade also possible:  a += b += c;
// (return reference to self; += is right associative)
Complex &Complex::operator+=(const Complex &rhs) {
    re += rhs.re;
    im += rhs.im;
    return *this;
}

// case: -a
Complex Complex::operator-() const {
    return Complex(-re, -im);
}
```

Pre/Post ++ - -

Distinguish `++i` from `i++`

For number types, both increment `i`, but the ***VALUE*** of both expressions is different:

- The value of `++i` (pre++) is the ***REFERENCE*** to the variable
- The value of `i++` (post++) is the ***VALUE*** of the variable ***BEFORE*** increment

Same for - -

```cpp
int i = 5, j = 5;

cout << (i++) ;      // writes 5, i == 6 after
cout << (++j) ;      // writes 6, j == 6 after

i++++;               // illegal because result of i++ is not a
                     // variable (a.k.a. lvalue)
++++i;               // OK, ++i returns a reference to i
```

Pre/Post ++ - -

In general, post-increment/decrement operators are slower, because they need to store the value of the object prior to incrementing/decrementing and return the copy.

Example:

```cpp
// pre++ : faster
Complex &operator++() {
    ++re;
    // return reference to current state
    return *this;
}

// post++ : slower
Complex operator++(int) {
    // memorize previous state
    Complex ret(*this);
    ++re;
    // return copy of previous state
    return ret;
}
```

Operator Overloading Tips

- Similar operators shall perform similar actions
  - `+=` `++` `+` should all deal with addition
  - `-=` `--` `-` should all deal with subtraction
- Use ***REFERENCE*** parameters whenever you can, but return ***VALUES*** when you must
  - Example: `T operator+(const T &rhs)`
  - There is no way around returning by value
  - T* doesn't work : `a + b + c` illegal
  - T& : reference to local variable (doesn't work) or object on heap : slow, and who is cleaning up? When evaluating `a + b + c` we don't have access to temporary variables, so we can't clean up even if we wanted
- Avoid complex expressions with side effects
  - If in doubt, break up expressions to enforce evaluation order

Operator Overloading Tips

Example:

```
x = x / ++x;  =>    y = x;  x = y / ++x;

y = f() + g(); (if f and g access global variables things can get tricky)

                =>    x = f(); y = x + g();
```

This works because `;` marks a so-called *sequence point*. At those points the C++ specification guarantees that all preceding code has been executed before execution continues after the sequence point.

Never overload the following

`&`   `&&`   `||`   `,`

because this certainly will confuse readers of your code including yourself! Recall that & takes the address of a variable, && and `||` are Boolean short-cut operators, and `,` is the sequence operator. Imagine what happens when a *clever* team member changes the meaning of those operators ...