Part 4: Type Casts, Static, Operator Overloading

Contents

[DOCUMENT NOT FINALIZED YET]

Type Casts

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

```
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)

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 (DON'T!)

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]);
}
```

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?

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:

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

private:
  mutable int get_n;
};
```

Because mutable variables can be changed in const functions, they mustn't be essential for the object state

dynamic_cast

Used for "walking up and down the type hierarchy"

```
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);

X *pyx = py;  // this is equivalent
```

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();
}
...
```

The presence of `dynamic_cast` usually indicates a broken class design. Use virtual functions instead

`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

Tip: try to avoid casting operations, but if you must use them, prefer C++ casts over `(T)`-style C casts, which are inherently unsafe

Static Data and Functions

Using "static" outside class definitions:

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.
}
```

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

  cout << get_x().a << endl; // use object
}
```

`static` in Class Context

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

Example

```
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;
};

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

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;
}
```

Operator 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;

cout << a;
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!

The following table gives an almost complete overview of C++ operators and almost all of the listed operators can be customized

C++ Operator Table

::	ltr 17 (high)
post++ post-- () [] . ->	ltr 16
! ~ pre++ pre-- + - * & (type) sizeof new new[] delete delete[]	rtl 15
.* ->*	ltr 14
* / %	ltr 13
+ -	ltr 12
<< >>	ltr 11
< <= > >=	ltr 10
== !=	ltr 9
&	ltr 8
^	ltr 7
|	ltr 6
&&	ltr 5
||	ltr 4
?:	rtl 3
= += -= *= /= %= &= |= ^= <<= >>=	rtl 2
,	ltr 1 (low)

- post = postfix operator (e.g., i–)
- pre = prefix operator (e.g., ++i)
- rtl: right (to left) associative, ltr: left (to right) associative
- cyan boxes: arity 1 (unary operators), all others arity 2 (binary), except for ?:
- number: precedence level (e.g., == binds tighter than =)

Examples:

`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)++`

This works because expressions are evaluated inside-out with respect to the parenthesis level. E.g. the atomic steps to evaluate `r = (a + b) * (c + d)` are

`x = a + b; y = c + d; r = x * y;`

Complex Number Example

```
// 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

```
cout << x << y;

cin >> x >> y;
```

Example:

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

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

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

// doesn't work: re,im are private and can't be accessed
// outside the class
```

Solution: Friends or Getters/Setters

```
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. E.g.

```
a + b    ->  a.operator+(b)

a += b   ->  a.operator+=(b)

v[2*i+1] ->   v.operator[](2*i+1)

f(x, y) ->  f.operator()(x, y)
            // f is called a "functor",
            // looks like a regular function call

++x      ->  x.operator++()

x++      ->  x.operator++(0)
            // 0 is a dummy parameter indicating
            // post increment
```

I.e.,

`T operator++() {...}` defines the prefix ++ operator, and `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 functions!

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()
```

int-Vector Revisited

```
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
```

Why Two Definitions of operator[] ?

```
class Foo
{
public:

  V a;
  ...
  int bar() const
  {
    return a[0];
    // Only works if const definition is provided
    // for V[]. Otherwise, the compiler complains
    // that bar() may change members of a. In
    // const contexts the const version is called
    // and in non-const cases the first version
    // is called.
  }
};
```

Complex Number Example Continued

```
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
};
```

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

```
#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 --

Example:

```
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
```

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:

```
// 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"
  etc.
- 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?

  Plus: 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
  The value of any expression involving more than one operation with side effects is undefined because evaluation order depends on the compiler and may affect the result

If in doubt, break up expressions to enforce evaluation order

Examples:

```
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 these points the C++ specification guarantees that all preceeding code has been executed before execution continues after the sequence point

- Never overload

  unary & && || ,

  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 these operators ...