

C++ and operator overloading

- in this lecture we will examine operator overloading
- how it is possible to introduce your own data type to the language seamlessly

C++ and operator overloading

- you can define almost all C++ operators for class or enumeration operands
 - called operator overloading
- we have already looked at
 - assignment (1 of the big three)
 - copy/delete are the other two

C++ and operator overloading

- while you can overload +, -, * and /
- it is often more useful to overload (), [], =, ==, !=, <, >, <= and >=

C++ and operator overloading

- generally it is not a good idea to define operators for a type unless you are really sure it adds clarity to your own code
- conventional wisdom among C++ programmers is that operators should only be overloaded with their conventional meaning
- only advice - there may occasionally be good reasons to break this, but be careful

Learning about overloading through an example

- let us build a fractional data type which takes the form: $whole + \frac{num}{denom}$
- the three values: whole, denom and num are defined as having the type long unsigned int
- the `fract` data type is useful as it allows us to retain absolute precision avoiding rounding errors

fract

c++/fractions/fract.h

```

#ifndef FRACTH
# define FRACTH

#include <iostream>

typedef long unsigned int longcard;

class fract
{
private:
    bool positive;
    bool top_heavy; // if true it _might_ be top heavy,
                  // false it is _not_ top heavy
    longcard whole;
    longcard num;
    longcard denom;
    fract not_top_heavy (void);
    bool is_top_heavy (void); // if true it _might_ be top heavy,
                             // false it is _not_ top heavy
    fract addND (fract right);
    bool subND (fract &left, fract right);
    friend std::ostream& operator<< (std::ostream& os, const fract& f);
};

```

fract

fract

c++/fractions/fract.h

```

public:
    fract (void);
    ~fract (void);
    fract (const fract &from);
    fract& operator= (const fract &from);
    fract (int);
    fract (int, int);
    fract (longcard w);
    fract (longcard n, longcard d);
    fract simplify (void);
    bool is_positive (void);
    bool is_negative (void);
    fract inc (fract right);
    fract dec (fract right);
    fract negate (void);
    bool is_zero (void);
    fract reciprocal (void);

```

c++/fractions/fract.h

```

fract operator+ (const fract &right); // fract + fract
fract operator+ (int right); // fract + int
friend fract operator+ (int left, const fract &right);

fract operator* (const fract &right); // fract * fract
fract operator* (int right); // fract * int
friend fract operator* (int left, const fract &right);

fract operator- (const fract &right); // fract - fract
fract operator- (int right); // fract - int
friend fract operator- (int left, const fract &right);
};

#endif

```

How fract might be used

c++/fractions/test-fract.cc

```
{
  fract a = fract (1, 2);
  std::cout << "a = " << a << "0;
}
```

c++/fractions/test-fract.cc

```
{
  fract a = fract (1, 2);
  fract b = fract (1, 2);
  fract c = a + b;

  std::cout << a << " + " << b << " = " << c << "0;
}
```

How fract might be used

c++/fractions/test-fract.cc

```
{
  fract a = fract (1, 2);
  fract b = a + 1;

  std::cout << a << " + " << 1 << " = " << b << "0;
}
{
  fract a = fract (1, 2);
  fract b = 1 + a;

  std::cout << 1 << " + " << a << " = " << b << "0;
}
```

Tutorial

- complete the operator* in the file
c++/fractions/fract.cc
- implement the appropriate methods for overloading
the / operator
- add some test code in c++/fractions/test-
fract.cc to test your / and * operators
- read c++/fractions/fract.cc and understand
it