Operator Overloading

User-Defined Operators and Conversions

Operator Overloading

- One of the goals of C++ is to allow used-defined types to be as convenient as built-in types

```
void fun1(double d1) {
  double d2 = 2.0;
  double input;

  cin >> input;

  cout << (d1 + d2 * input);
}
```

convenient

```
void fun2(Complex c1) {
  Complex c2(3.0, 2.0);
  Complex input;
  input.readFrom(cin);
  c2.multiply(input);
  c1.add(c2);
  c2.printTo(cout);
}
```

annoying

- For that purpose, C++ allows defining **operators** (such as + or =) for user-defined types

```
void fun2(Complex c1) {
  Complex c2(3.0, 2.0);
  Complex input;

  cin >> input;

  cout << (c1 + c2 * input);
}
```

awesome!

Operator Overloading

- Defining operators is done using **operator overloading**
  - C++ treats an operator like a function
  - Operators can be **overloaded** the same way functions can

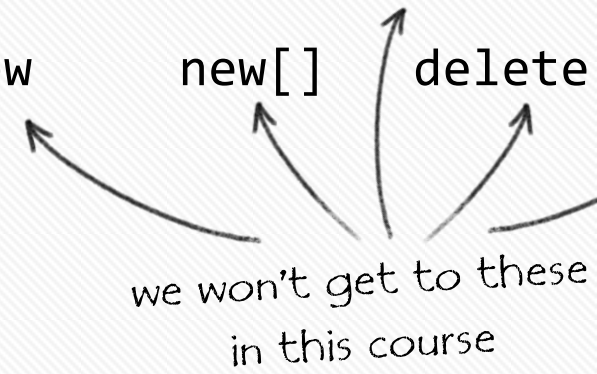
operator+ is just a function name
(incidentally, the function called for the operator '+')

```
Complex operator+(const Complex& x, const Complex& y) {
  Complex result(x.real() + y.real(), x.imag() + y.imag());
  return result;
}
```

What Can We Do?

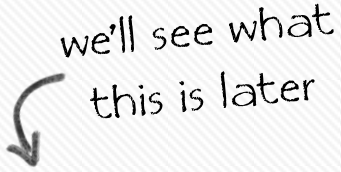
- We can overload all of the following operators:

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]



What Can't We Do?

- We cannot:
  - Overload these operators:
    `::` `.` `.*` `? :` `sizeof` `typeid`



  - Define **new operators**
    - So "a**b" for a power operator cannot be defined
  - Change the **number of parameters**, **precedence** or **associativity** of existing operators
    - So "`a + b * c`" always means "`a + (b * c)`"
  - An overloaded operator must have **at least one parameter** which is a **user-defined type** (a class, struct or enum)

What Should We Do?

- Operator overloading should be used to **simplify code**
  - Not make it **cryptic**

bad ideas

```
list += element; // add element to list
list--; // remove last element from list
```

- We should only use operator overloading when the behavior of the operator is **natural and intuitive**
  - Otherwise, we should just use normal functions with **meaningful names**

much better

```
list.addFirst(element);
list.removeLast();
```

- We want our overloaded operators to behave **similarly to their built-in counterparts**
  - So **+= for complex** numbers should behave like **+= for doubles** :

```
double d1 = 2.0;
d1 = d1 + 1.0;
```

equivalent

```
double d1 = 2.0;
d1 += 1.0;
```

same equivalence should be true for overloaded operators

Binary Operators

- A binary operator can be defined in one of two ways:
  - As a **member** function taking **one argument**
  - As a **nonmember** function taking **two arguments**

- For any binary operator **@**, the compiler interprets **a@b** as either **a.operator@(b)** or **operator@(a,b)**

```
class Complex {
  double re, im;
public:
  //...
  Complex operator+(const Complex& x) const;
};
```

binary operator+ declared as member

or

binary operator+ declared as nonmember

```
Complex operator+(const Complex& x, const Complex& y);
```

Unary Operators

- A unary operator can be defined in one of two ways:
  - As a **member** function taking **no arguments**
  - As a **nonmember** function taking **one argument**

- For any prefix unary operator **@**, the compiler interprets **@a** as either **a.operator@()** or **operator@(a)**
  - The *postfix* ++ and -- operators are treated a bit differently (more later)

```
class Complex {
  double re, im;
public:
  //...
  Complex operator-() const;
};
```

a member unary operator-

```
Complex operator-(const Complex& x);
```

a nonmember unary operator-

Other Operators

- The following operators must be overloaded **as member functions**:

```
=       []      ()      ->
```

- The `->` and `()` operators have further rules:
  - `()` can be overloaded with **any number of parameters**
  - `->` must return an object for which `->` is a valid operator (i.e., a pointer or some class which defines operator`->`)
    - We will see examples for both of these later

A Rational Number Class

- A **rational number** is a number that can be expressed as a fraction a/b of two integers

- We will use operator overloading to create a rational number class that behaves **similarly to a built-in type**
  - We will keep the number in simplified form
    - The **numerator** and **denominator** will be co-prime
    - The **denominator** will be a positive number
  - The only mathematical background needed for this class is:
    - Acquaintance with the Euclidean algorithm for finding the **GCD** (greatest common divisor) of two positive integers

A Rational Number Class

converts this object to simplified form

```
class Rational {
  int numerator, denominator;

  void simplify();
  static int gcd(int x, int y);
public:
  Rational(int numerator = 0,
           int denominator = 1);
  //... functions + operators ...
};
```

```
void Rational::simplify() {
  if (denominator < 0) {
    denominator = -denominator;
    numerator = -numerator;
  }

  int common = gcd(numerator, denominator);
  if (common != 0) {
    numerator /= common;
    denominator /= common;
  }
}
```

we'll get back to this when we learn exceptions

```
Rational::Rational(int numerator, int denominator) :
  numerator(numerator), denominator(denominator) {

  if (denominator == 0) {
    throw DivisionByZero();
  }

  simplify();
}
```

Operators + and +=

- Let us start by overloading operator+ and operator +=
- What should the declaration for each of those be?
  - To answer that, we need to look at how operator+ and operator+= behave for a built-in type

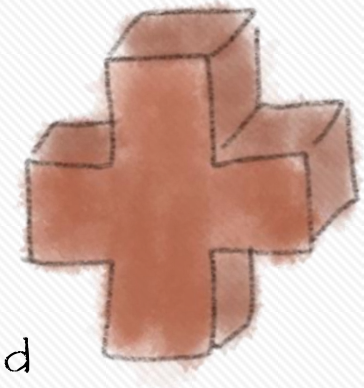
```
int a = 1;
const int b = 2;

cout << (a + b);
cout << (b + 3);

int& c = a + b; // error
```

returns a temporary value

both sides are unchanged



```
class Rational {
  //...
  Rational operator+(const Rational& r) const;
};
```

Operators + and +=

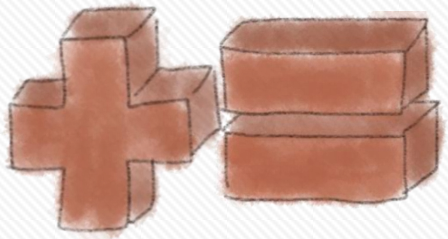
- Let us start by overloading operator+ and operator +=
- What should the declaration for each of those be?
  - To answer that, we need to look at how operator+ and operator+= behave for a built-in type

```
int a = 1;
const int b = 2;

cout << (a += b);
cout << (a += 3);

(a += b) += 2;
b += 2; // error
```

return value is not a temporary!
it's the first argument itself



right-hand side
is unchanged

left-hand side is not const

```
class Rational {
  //...
  Rational& operator+=(const Rational& r);
};
```

Operators + and +=

- We can now **implement** our operators, however:
  - It seems + and += should have a similar implementation
  - How can we avoid **code duplication**?
- Somewhat surprisingly, **+ is more complex to implement than +=**
  - Operator+ involves 3 objects, while += only involves 2
  - Therefore, **we implement + using +=**

```
Rational& Rational::operator+=(const Rational& r) {
  numerator = numerator * r.denominator + r.numerator * denominator;
  denominator = denominator * r.denominator;
  simplify();
  return *this;
}
```

```
Rational Rational::operator+(const Rational& r) const {
  Rational result(*this);
  return (result += r);
}
```

A Rational Number Class

- Operators such as * and *= are implemented similarly
- Operator - (minus) has **two versions**:
  - A **binary** operator- , as in "a - b "
  - A **unary** operator- , as in "-a "

```
class Rational {
  //...
  Rational& operator-=(const Rational& r);
  Rational operator-(const Rational& r) const;
  Rational operator-() const;
};
```

```
Rational Rational::operator-() const {
  return Rational(-numerator, denominator);
}
```

```
Rational& Rational::operator-=(const Rational& r) {
  return *this += -r;
}
```

```
Rational Rational::operator-(const Rational& r) const {
  return Rational(*this) -= r;
}
```

Type Conversions

- C++ allows **user-defined conversions** between types
  - At least one of the types in the conversion must be user-defined
  - If a conversion is available, the compiler can **implicitly cast function arguments** when needed

```
Rational r1(1, 2);
int n = 1;
Rational r2 = r1 + n;
```

'n' is implicitly cast from int to Rational (=n/1)

- Conversions can be defined in two ways:
  - By a **constructor** which takes one argument
  - By overloading a **conversion operator**

Conversion by Constructor

- A constructor which **takes a single argument** defines a conversion:

```
Rational::Rational(int numerator) :
  numerator(numerator), denominator(1) {
  simplify();
}
```

convert an int into a Rational

- A constructor that can be called with a single argument using **default values** also defines a conversion:

```
Rational::Rational(int numerator, int denominator = 1) :
  numerator(numerator), denominator(denominator) {

  if (denominator == 0) {
    throw DivisionByZero();
  }

  simplify();
}
```

Conversion by Constructor

- Let's analyze the following call to operator+:

```
int n = 2;
Rational r2 = r1 + n;
```

1. Compiler searches for an exact match: `Rational::operator+(int)`

no such function exists

2. Compiler searches for a function of the form `Rational::operator+(T)` where `T` is a type that **int can be converted to**

int → Rational conversion is available via `Rational::Rational(int)`

3. Compiler uses the function `Rational::operator+(const Rational&)`

code is implicitly changed to

```
Rational r2 = r1 + Rational(n);
```

Conversion Operator

- Sometimes we will want to create a conversion **from a new type to an existing type**
  - We cannot always modify the existing type (it may be built-in!)
- A special conversion operator may be defined with the following syntax:

no return type (it is taken from the operator's name)

```
class Rational {
  //...
  operator double() const;
};
```

```
Rational::operator double() const {
  return double(numerator) / denominator;
}
```

- One typical use of this feature is to define a conversion to type **bool**, so objects can be used as conditions:

converts to true if all input operations completed successfully

```
int input;
cin >> input;
if (cin) {
  cout << "number read successfully";
}
```

Explicit Conversions

- **Implicit conversions** are usually a bad idea
  - Code might become **ambiguous**:

Ambiguous code does not compile

```
class Rational {
  //...
  Rational(double val);
  operator double() const;
};
```

Just for the example

```
Rational r(1, 2);
cout << r + 1.2;
```

Compiler can't decide between:
1. convert 1.2 to Rational
2. convert r to double

  - Nasty **bugs will compile**:

```
class Array {
  // ...
  Array(int size);
};
```

```
void f(Array& a, int i) {
  a = 5; // converts 5 to a temporary empty array
         // of length 5, and assigns it to a.
         // programmer probably meant a[i] = 5;
}
```

Explicit Conversions

- To **prevent the compiler** from converting values automatically via a constructor or conversion operator, we can declare them as **explicit**
  - Explicit **conversion operators** have been added in C++11

```
class Array {
  //...
  explicit Array(int size);
};
```

```
class Rational {
  //...
  Rational(int num = 0, int denom = 1);
  explicit operator double() const;
};
```

C++11

```
void f(Array& a, int n) {
  a = n;
}
```

does not compile anymore. if you really meant it, use

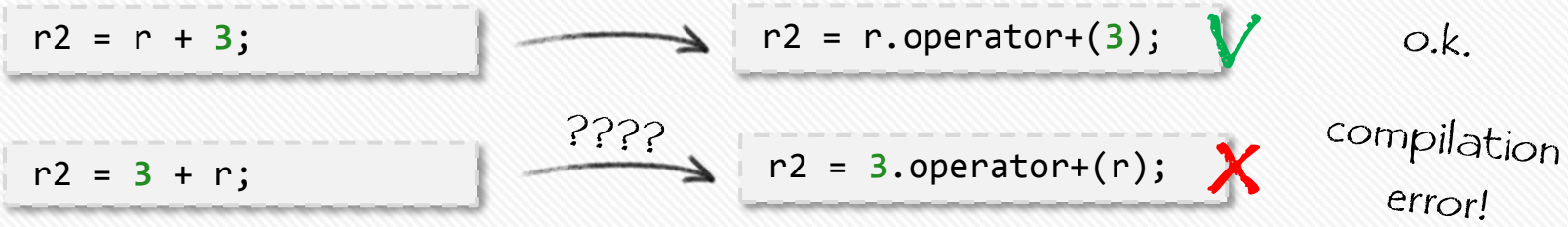
```
void f(Array& a, int n) {
  a = Array(n);
}
```

```
Rational r2 = r1 + 1;  // still compiles - Rational(int) is not explicit
double x = cos(r1);    // will not compile (good!), must use cos((double)r1)
```

- **Good rule of thumb**: always declare a constructor taking a single parameter as explicit, unless a conversion really makes sense

Conversions and Member Functions

- Our operator+ has one problem…



- This means that our + operator for Rationals has **asymmetrical behavior**
  - We cannot switch addition order!
- To solve this, we define operator+ as a **non-member** (ordinary) function:

```
Rational operator+(const Rational& r1, const Rational& r2) {
   Rational result = r1;
   result += r2;
   return result;
}
```

- Now **both sides** of the + can be converted if necessary, not just the left side

Friends

- Lets add an **== operator**:
  - operator== should return a bool
  - operator== should not change any of its arguments
  - We would like to have symmetric behavior, so it should be a **non-**

```
bool operator==(const Rational& r1, const Rational& r2) {
  return r1.numerator == r2.numerator && r1.denominator == r2.denominator;
}
```

Oops! Need to access private members

- Using a **friend declaration** we can grant an **external function** access to the private section of a class

```
class Rational {
  //...
  friend bool operator==(const Rational& r1, const Rational& r2);
};
```

Friends

- A friend declaration can appear **in any part of the class** (public or private) – it doesn't matter
  - In practice, it makes more sense to place it in the private section unless the user of the class should be aware of it

```
class Rational {
public:
  friend void f(const Rational&);
};
```

Equivalent

```
class Rational {
private:
  friend void f(const Rational&);
};
```

- Friend declarations actually perform **two tasks** –
  - They grant private access to the external function
  - They declare the existence of the external function, so that a separate declaration is not necessary*

*Actually, this only applies when the friend function takes at least one parameter of the class type, but this will always be the case anyway

Overloading Input/Output Operators

- The **<<** and **>>** operators are used in C++ for **I/O operations**
  - They can be overloaded to **support input and output of user-defined types**
- Both operators must be defined as **non-members**
  - Their first argument is an I/O stream and not an object of the class

```
std::ostream& operator<<(std::ostream& os, const Rational& r) {
  os << r.numerator;
  if (r.denominator != 1) {
    os << "/" << r.denominator;
  }
  return os;
}
```

```
class Rational {
  // ...
  friend std::ostream& operator<<(std::ostream& os, const Rational& r);
};
```

The Final Rational Class (1/2)

rational.h

```
class Rational {
  int numerator, denominator;
  void simplify();
  static int gcd(int x, int y);

public:

  Rational(int numerator = 0, int denominator = 1);
  explicit operator double() const;

  Rational operator-() const;
  Rational& operator+=(const Rational&);
  Rational& operator-=(const Rational&);
  Rational& operator*=(const Rational&);
  Rational& operator/=(const Rational&);

  friend bool operator==(const Rational&, const Rational&);
  friend bool operator<(const Rational&, const Rational&);
  friend std::ostream& operator<<(std::ostream&, const Rational&);
};

...
```

The Final Rational Class (2/2)

rational.h

```
...

Rational operator+(const Rational&, const Rational&);
Rational operator-(const Rational&, const Rational&);
Rational operator*(const Rational&, const Rational&);
Rational operator/(const Rational&, const Rational&);

bool operator!=(const Rational&, const Rational&);
bool operator>=(const Rational&, const Rational&);
bool operator>(const Rational&, const Rational&);
bool operator<=(const Rational&, const Rational&);
```

all of these can be implemented outside the class using only its interface

avoid using friend declarations when not needed

Assignment Operator

- An assignment operator= is automatically generated by the compiler
  - The compiler-generated operator= calls operator= for each of the members of the class, i.e.

```
Rational& Rational::operator=(const Rational& r) {
  numerator = r.numerator;
  denominator = r.denominator;
  return *this;
}
```

compiler generated

  - In some cases, the compiler-generated operator= **will not work** as intended
    - Usually happens when the class manages a resource

Uh-oh! Both sets now share the same array

```
Set& Set::operator=(const Set& s) {
  size = s.size;
  maxSize = s.maxSize;
  data = s.data;
  return *this;
}
```

compiler generated

Assignment Operator

- We can **overload the assignment operator** of a class
  - This should be done only when the compiler-generated one is insufficient
  - An assignment operator should **release old resources**, and **allocate new ones**
  - We should watch out for the case of a self assignment

```
Set& Set::operator=(const Set& s) {
  if (this == &s) {
    return *this;
  }
  delete[] data;
  data = new int[s.size];
  size = s.size;
  maxSize = s.size;
  for (int i = 0; i < size; i++) {
     data[i] = s.data[i]
  }
  return *this;
}
```

avoids a bug in case of self-assignment

```
Set s1;
// ...
s1 = s1;
```

The Big Three

- These three functions typically **go together**:
  - Copy constructor
  - Destructor
  - Operator=
- If you need to write **one**, you probably **need all three**
- Intuitive reasoning:

Assignment = Destructor + Copy C'tor

Control of default class methods

- We have seen that the compiler automatically generates several methods for us. For example:
  - A default constructor is generated whenever no other constructor is explicitly defined
  - A copy constructor and assignment operator are **always** generated if the user dies not define them explicitly
- This behavior **may not always be what we want**

```
class Person {
  std::string name;
  // …
public:
  Person(std::string name);
};
```

We want the default c'tor, but it will not be generated for us

```
class ImageWriter {
  // …
public:
  ImageWriter(std::string filename);
  void write(Image im);
};
```

We do not want this class to be copyable or assignable, but the compiler generates these anyway

Control of default class methods

- We can use **=default** to direct the compiler to generate a method that it would not normally generate

C++11

```
class Person {
  std::string name;
  // …
public:
  Person() = default;
  Person(std::string name);
};
```

Tells the compiler to generate the default c'tor even though another one is defined

- Using =default is much better than **implementing the default behavior ourselves**

For example, whenever fields are added or removed from the class, we will have to remember to update this function manually or else something will break

Control of default class methods

- What if we don't want our class to be copyable or assignable at all?
  - In some cases there is **no reasonable meaning** for these operations

C++98

```
class ImageWriter {
  // …
private:
  ImageWriter(const ImageWriter&);
  void operator=(const ImageWriter&);
};
```

Trying to copy or assign from outside the class will be a compiler error, but from within the class it will only be a link error

C++11

```
class ImageWriter {
  // …
public:
  ImageWriter(const ImageWriter&) = delete;
  void operator=(const ImageWriter&) = delete;
};
```

Trying to copy or assign will always be a compiler error. Also, we can place the =delete in the public section as part of the interface.

Control of default class methods

- We can also use **=default** to emphasize that we did not forget to implement a method

```
class Image {
  // …
public:
  Image(int width, int height);
  int getPixel(int x, int y) const;
};
```

The compiler will generate a copy c'tor and operator= for this class. But, did the programmer really **mean for this**, or did he just **forget** to implement them?

```
Image(const Image&) = default;
Image& operator=(const Image&) = default;
```

Adding these to the public section is the best way to **indicate our meaning**

- Always indicate constructors and assignment operators with **=default** if you want the compiler to generate their default behavior

Except for simple structs

Indexing Operator

- The **indexing operator []** can be overloaded to create classes that behave like arrays
  - The parameter does not have to be of type int
- The indexing operator **must be defined as a member function**

```
class Array {
  int* data;
  int size;
public:
  Array(int n);
  Array(const Array& a);
  ~Array();
  Array& operator=(const Array& a);
  int size() const;
  int& operator[](int index);
};
```

```
int& Array::operator[](int index) {
  assert(index >= 0 && index < size);
  return data[index];
}
```

```
Array& doubleIt(Array& a) {
  for (int i = 0; i < a.size(); i++) {
    a[i] = 2 * a[i];
  }
  return a;
}
```

Indexing Operator

- Functions which **return a reference**, such as our [], pose a problem:

the call a[i] will not compile since operator[] is not const

```
Array arrayDouble(const Array& a) {
  Array result(a.size());
  for (int i = 0; i < a.size(); i++) {
    result[i] = 2 * a[i];
  }
  return result;
}
```

- We will need **two versions** of operator[]:
  - One version to call for **const** objects
  - One version to call for **regular** objects

make sure this function actually protects the object from changing

```
class Array {
  int* data;
  int size;
public:
  // ...
  const int& operator[](int index) const;
  int& operator[](int index);
};
```

```
int& Array::operator[](int index) {
  assert(index >= 0 && index < size);
  return data[index];
}
```

```
const int& Array::operator[](int index) const {
  assert(index >= 0 && index < size);
  return data[index];
}
```

Iterators

- We still need to add support for **enumerating** our integer set
- A simple solution could be to **use pointers**
  - **Natural** usage syntax
  - **Multiple iterators** to the same set can co-exist

```
class Set {
  // ...
public:
  // ...
  const int* begin() const;
  const int* end() const;
};
```

```
const int* Set::begin() const {
  return data;
}
```

```
const int* Set::end() const {
  return data + size;
}
```

pointer to the "end" of the array

```
void sumOfSet(const Set& set) {
  int sum = 0;
  for (const int* i = set.begin(); i != set.end(); ++i) {
    sum += *i;
  }
  cout << "sum of set is " << sum << endl;
}
```

using the iterator

Iterators

- Using pointers as iterators is **implementation-dependent**
  - What if our set were implemented with a linked list?
- However, the resulting syntax from using pointers is **simple** and **clean**
- Using operator overloading we can enjoy the **best of both worlds**
  - We can **create sophisticated iterators** which will be controlled through the **same simple interface as pointers**

Iterators

- As an example, let us add the following **feature** to our set's iterator:
  - Assert that the iterator is dereferenced with operator* **only when pointing to a valid element**

```
void undefined_fcn(const Set& set) {
  const int* i = set.end();
  cout << *i << endl; // undefined!
}
```

we prefer a **detailed error message** in such a case

- To achieve this, we must replace the simple pointer iterator with a **class** which we will implement

```
class Set {
public:
  // ...
  class Iterator;
  Iterator begin() const;
  Iterator end() const;
private:
  // ...
};
```

note that Set::Iterator is public, and thus part of the interface of Set

declare a class Set::Iterator

yes, we can do that

Minimal Operators for an Iterator

- Any iterator class in C++ must support **at least** the following three operators:

operator!=()
to compare two iterators

operator*()
to return the object currently pointed to

```
void sumOfSet(const Set& s) {
  int sum = 0;
  for (Set::Iterator it = s.begin(); it != s.end(); ++it) {
    sum += *it;
  }
  cout << "sum of set is " << sum << endl;
}
```

prefix operator++()
to advance the iterator

Optional Operators for an Iterator

- In addition, we may wish to support **all or some** of the following operators, which mimic the behavior of pointers:

operator==()
for comparing iterators

postfix operator++()
for advancing the iterator

copy c'tor and operator=()
(note that it2 initially points to the same location as it1, but it moves independently)

prefix/postfix operator--()
for moving the iterator backwards

```
void func(const Set& s) {
  Set::Iterator it1 = s.begin();
  if (it1 == s.end()) {
    return;
  }
  cout << "first = " << *it1 << endl;
  it1++;
  Set::Iterator it2 = it1;
  cout << "second = " << *it2 << endl;
  it1--;
  cout << "first again = " << *it1 << endl;
  cout << "second again = " << *it2 << endl;
}
```

iterators that supports this are called a bidirectional iterators
(we will not implement these here)

Set::Iterator Interface

set.h

```
class Set {
public:
  // ...
  class Iterator;

  Iterator begin() const;
  Iterator end() const;

private:
  int* data;
  int size;
  int maxSize;
  // ...
};
```

this appears after the declaration of Set::Iterator

later in set.h

```
class Set::Iterator {
  const Set* set;
  int index;
  Iterator(const Set* set, int index);
  friend class Set;
public:
  const int& operator*() const;
  Iterator& operator++();
  Iterator operator++(int);

  bool operator==(const Iterator& it) const;
  bool operator!=(const Iterator& it) const;

  Iterator(const Iterator&) = default;
  Iterator& operator=(const Iterator&) = default;
};
```

Set::Iterator Interface

why is the c'tor private?

so no one except Set can create a Set::Iterator (careless invocations of this c'tor can break the set)

```
class Set::Iterator {
  const Set* set;    // the set this iterator points to
  int index;         // the current index in the set
  Iterator(const Set* set, int index);
  friend class Set;  // allow Set to call the c'tor
public:
  const int& operator*() const;
  Iterator& operator++();    // prefix (++it)
  Iterator operator++(int);  // postfix (it++)
  // ...
};
```

Set::Iterator Interface

Set may access the private members of Set::Iterator

allows Set to call Iterator's private constructor (Set::begin() and Set::end() require this to create an Iterator)

```
class Set::Iterator {
  const Set* set;    // the set this iterator points to
  int index;         // the current index in the set
  Iterator(const Set* set, int index);
  friend class Set;  // allow Set to call the c'tor
public:
  const int& operator*() const;
  Iterator& operator++();    // prefix ++
  Iterator operator++(int);  // postfix ++
  // ...
};
```

*In some cases, the usage of friends may indicate bad design. However, don't replace a friend declaration by simply making things public

Implementing Set::Iterator

set.cpp

```
Set::Iterator::Iterator(const Set* set, int index) :
  set(set),
  index(index)
{ }
```

```
const int& Set::Iterator::operator*() const {
  assert(index >= 0 && index < set->getSize());
  return set->data[index];
}
```

Set::Iterator is inside the Set class, so it has access to the private section of Set

Implementing Set::Iterator

Note the difference in return type of ++i and i++

```
Set::Iterator& Set::Iterator::operator++() {
  ++index;
  return *this;
}
```

```
Set::Iterator Set::Iterator::operator++(int) {
  Iterator result = *this;
  ++*this;
  return result;
}
```

the "dummy" int parameter tells the compiler this is a postfix ++

Implementing Set::Iterator

comparing iterators of two different sets is a bug

```
bool Set::Iterator::operator==(const Iterator& i) const {
  assert(set == i.set);
  return index == i.index;
}
```

```
bool Set::Iterator::operator!=(const Iterator& i) const {
  return !(*this == i);
}
```

Implementing Set::Iterator

Back in class Set, we need to implement the functions that actually return the iterators

```
Set::Iterator Set::begin() const {
  return Iterator(this, 0);
}
```

```
Set::Iterator Set::end() const {
  return Iterator(this, size);
}
```

“The problem with using C++ is that there's a strong tendency in the language to require you to know everything before you can do anything.”

- Larry Wall