Chapter

26

More on Classes

A working constructor is the short-hand description of a constructor that takes as many user-supplied arguments as possible. In the `Time` class the working constructor looks like this:

```
Time(int hours, int minutes);
```

In the `.cpp` file, you might have code that looks something like this, using the initializer list from the last lesson:

```
Time::Time(int hours, int minutes)
   : m_hours(hours), m_minutes(minutes)
{ }
```

Conversion Constructors

A **conversion constructor** is a constructor (usually 1-argument) that implicitly converts between one type and another. Here's an overloaded conversion constructor that converts between fractional hours and hours and minutes:

```
Time(double hours);  // 7.51 -> 7:35
```

The implementation of the constructor (converting first to seconds, then extracting the hours and minutes) could look like this:

```
auto time = static_cast<long long>(hours * 3600);
m_hours = time / 3600;
m_minutes = time % 3600 / 60;
```

Conversion constructors can be **implicit** or **explicit**. The implicit conversion constructor is called any time the compiler needs a `Time` object, but finds a `double` that it can convert. Consider this fragment of code:

```
Time bedTime(23, 30);    // 11:30 pm
```

```
bedTime = 5.2;                // WHAAAAT?
```

You would **expect** that the highlighted line would be a syntax error, but, surprisingly, it is not. Instead, **the conversion constructor is silently called**, and `bedTime` is changed to `5:12 am`. Probably not what you expected.

Add `explicit` as a modifier to the prototype to prevent this.

```
explicit Time(double hours);  // 7.51 -> 7:35
```

The keyword `explicit` only goes in the class definition. It **is not repeated** in the `.cpp` file. Sometimes, as we'll see when we cover symmetric overloaded operators, you'll **want** to allow implicit conversion. For instance the `string(const char *)` constructor is **not explicit**. Most of the time, however, `explicit` is preferred.

Assignment and Copy Constructor

You may **assign one object to another**, just as you can assign one `int` variable to another, even though the data members are private. As with the built-in types, the result is a **copy** of the data in the original members. With class types, this is carried out by the overloaded **assignment operator**.

```
Time& operator=(const Time& rhs);
```

When you pass an object by value, or **initialize** a new object variable with another, instead of the assignment operator, the **copy constructor** is called:

```
Time(const Time& rhs);
```

C++ automatically writes a **synthesized assignment operator** and a **synthesized copy constructor** that work for simple types such as those in this course. In CS 250 you'll learn how to write your own assignment operators and copy constructors to create more dynamic types. You can **prevent** pass by value or assignment by adding the following to the definitions.

```
Time& operator=(const Time& rhs) = delete;
Time(const Time& rhs) = delete;
```

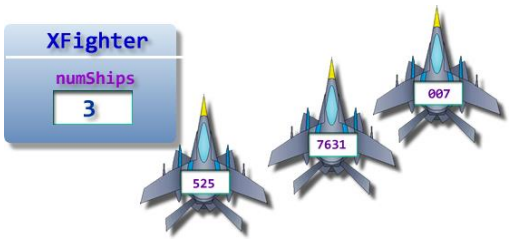
Destructors

When an object is **destroyed**, its **destructor** is called. If you don't write one, C++ writes one for you. If your class allocates dynamic memory, for instance, you would free it in the destructor. The destructor looks like the default constructor preceded by the **tilde**:

```
~Time();
```

const & static Members

Suppose you are creating a space-wars type video game, and one of your player types is an **XFighter** class. How do you easily keep track of **how many XFighter**s are currently available?



That's easy: with a **static**, or **shared data member** as so:

```
class XFighter
{
    static int numShips;
    int VIN;
public:
    XFighter(int n) : VIN(n) { numShips++; }
    ~XFighter { numShips--; }
};
```

The **static** member **numShips** is **private** to the **XFighter** class, but there is **only one copy** of the member, **not one for each ship**, like the vehicle identification number (**VIN**). When an **XFighter** is constructed, the constructor increments the shared **numShips**, and when a ship is destroyed, decrement it.

There is one wrinkle with this, that is different than in Java. In C++, you must **initialize the static member** as a global object in your **.cpp** file with:

```
int XFighter::numShips = 0;
```

This cannot go in the header file.

static Member Functions

How do you use the **numShips** variable? That is, how do **you** find out how many ships exist? With a **static member function**, like this:

```
static int xFighters() { return numShips; }
```

A **static** member function can only access **static** data members, or other **static** member functions. It cannot access regular data members. If the function is defined **outside of the header**, then you **do not** repeat the keyword **static**:

```
int XFighter::xFighters() { return numShips; }
```

Place **static** member functions in the **public** section of your class and call them using the class name and the scope operator, not an instance and the dot operator.

```
cout << XFighter::xFighters() << endl;
```

static const Data Members

When you have a constant that only applies to the situation represented by your class you can add it as a **static const** data member.

```
class Bizarro
{
public:
    static const int ANSWER{-42};
    static constexpr double E{3.14159};
    static const double PI;
};
```

In the Bizarro world, almost **everything is backwards**. The "answer to everything" is **-42**, not **42** as in our world. The mathematical constant **E** is **3.14159**, and the constant **PI** is the square root of **PI** in our own world.

For **integral types**, you may initialize **static const** data members inside the class; no other initialization is required. **Starting in C++11** you can also do this for other types, using the keyword **constexpr**, instead of **const**. However, if a type needs to calculate a value at runtime, (as **PI** does), you'll still need to provide a separate definition in the **.cpp** file.

```
const double Bizarro::PI = sqrt(acos(-1.0));
```

The data members are **static** (there is **only one copy** stored), they are **const**, (they **cannot be changed**), and they are **public** (you can use them outside of the class.)

```
cout << Bizarro::ANSWER << endl;
cout << Bizarro::PI << endl;
cout << Bizarro::E << endl;
```

Overloaded Operators

In Java, to compare two user-defined objects for equality, you override the inherited `equals()` method. Similarly, to add two `BigInteger` objects, you must use method-call syntax like this:

```
BigInteger a = new BigInteger("123");
BigInteger b = new BigInteger("50");

// Add b to a and return the result
BigInteger c = a.add(b);
```

With C++, (after creating the `BigInteger` class), you can write this instead.

```
BigInteger a{"123"};
BigInteger b{"50"};
auto c = a + b;
```

This is made possible by the C++ feature called operator overloading.

How Operator Overloading Works

```
bool Time::equals(const Time& lhs, const Time& rhs)
{
    return lhs.m_hours == rhs.m_hours &&
        lhs.m_minutes == rhs.m_minutes;
}
```

To compare two `Time`, structures, you could use the member function `equals()`. To convert this to an overloaded operator, simply change the name of the function from `equals` to `operator==`. That's all there is to it.

Here are the equivalent overloaded operators for equals:

```
bool Time::operator==(const Time& lhs, const Time& rhs)
{
    return lhs.m_hours == rhs.m_hours &&
        lhs.m_minutes == rhs.m_minutes;
}
```

And here is the equivalent operator for not-equals. Note that it is defined in terms of the previously written operator. You'll often write these operators in pairs like this.

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

The code for the `equals()` function and `operator==()` are the same.

- The compiler sees an expression like: `a = b + c`
- If `b` and `c` are built-in types (or convertible), then it uses the built-in addition
- If `b` or `c` are user-defined, it looks for an `operator+()` and calls that function

Syntax Rules

Overloaded operators have "special" names: the keyword `operator` followed by the symbol for the operator being defined. Here is a short list of syntax rules:

- You **may not** overload operators for the built-in types. **One operand must** be a user-defined type.
- You can only overload **existing operators**. (You can not define **.)
- You may not change the **precedence** or **arity** of an operator. A unary operator has one parameter, a binary operator two.
- You **may not** overload the **conditional** operator, the **scope** operator or the **member selection** operator.
- You **should not** overload the **comma**, **address**, `&&` and `||` operators.

In addition to the syntax rules, follow these guidelines:

- Operators should act **the same as, or similar to** the built-in operators. Using binary + for string concatenation is fine; using it for subtraction is not.
- **Define the I/O operators** (`<<` and `>>`) so they are compatible with how I/O is done for the built-in types
- If objects may be compared for equality, define `==` and `!=`.
- If objects have a single **natural ordering**, define `<` and most likely the other relational operators as well.
- The return type of an overloaded operator **should be compatible** with the return types of the built-in operators.

Most of these rules can be boiled down to: **When in doubt, do like the ints do**!

Operator Mechanics

There are two ways to overload operators on user defined types:

- **As member functions** (declared **inside the class**)

- **As non-member functions** (like those we created for structures.)

Non-Member Operators

As **non-member functions**, follow this pattern:

Syntax: Non-Member Overloaded Operator

```
struct Point { int x, int y; };
Point operator+(const Point& lhs, const Point& rhs);
double operator-(const Point& lhs, const Point& rhs);
Point operator-(const Point& obj);
Point& operator*=(Point& lhs, double scale);
```

1. Return type
2. Keyword operator and symbol
3. Parameters (unary: one, binary: two)

- The parameter names **lhs** and **rhs** are common when overloading a binary operator; they stand for left-hand-side and right-hand-side respectively.
- Non-member operators have **no direct access** to **private** members.
- The return type depends on the operator

Member Overloaded Operators

As **member overloaded operators**, follow this pattern:

Syntax: Overloaded Operator Member Functions

```
class Point
{
public:
   Point operator+(const Point& rhs) const;
   double operator-(const Point& rhs) const;
   Point operator*(double scale) const;
};
```

1. Return type
2. Keyword operator and symbol
3. Parameters (unary: none, binary: one)

Unlike the non-member version, there is no **explicit left-hand-side** parameter. The C++ **implicit parameter** (AKA **this**) is passed as the left-hand-side operator. That means a binary overloaded **member** operator **takes only one explicit parameter**, with the first parameter being passed **implicitly**.

The **addition operator** returns a new **Point** object[1] (by adding the member variables from the implicit and explicit argument). The **subtraction operator** finds the distance between two points (a **double** value), while the **multiplication operator** scales a point up or down by some factor.

Member or Non-Member?

Here are some guidelines (from Lipmann's C++ Primer) on how to decide whether an operator should be a member or a non-member. I have shortened them to only apply to operators we will cover in this class. You'll cover some of the others in CS 250.

- The **subscript operator** **[]** must be a member
- **Side-effect operators** (short-hand assignment, increment and decrement) should be members.

Symetric operators (those that may **convert either operand**) such as arithmetic, equality, and relational operators should be non-members.

[1] As you'll see later, it should actually return a const Point object.

Side-effect Operators

Side-effect operators are those that change (mutate) the object's state when called. This includes the assignment operator, shorthand assignment and the increment and decrement operators.

To see how this works, let's start with some examples, using this simple class:

```
class Int
{
    int m_val{0};   // private
public:
    // public members here
};
```

Arithmetic (side-effect) assignment operators (`+=` etc) modify their left-hand operand, and return a reference to that operand. They are usually written as **member operators**, so that they have direct access to the private data members.

Here is `operator+=()`.

```
Int& Int::operator+=(const Int& rhs)
{
    m_val += rhs.m_val;
    return *this;
}
```

1. As a **member operator**, the left-hand operand is the **implicit parameter**, `this`. The operator is not `const`, because `m_val` is modified.
2. The right-hand parameter **is `const`**. **It is not changed**.
3. The operator returns a **self-reference**: `*this`.

Increment and Decrement

The increment (and decrement) operators both have **prefix** and **postfix** versions.

- **Prefix** modifies the object and returns a reference to it (an ***lvalue***).
- **Postfix** saves the object, and returns a copy of that saved state as an ***rvalue***.

Here is the prefix operator for **Int**:

```
Int& Int::operator++()
{
    ++m_val;
    return *this;
}
```

Since both prefix and postfix are **unary operators**, and since both take the same number and type of arguments, **the compiler cannot distinguish them by using the signature**.

To handle this situation, the designers of C++ add an **extra, dummy parameter to the postfix** version, just to differentiate beween them. **Nothing is actually passed, and no name is supplied for the dummy parameter in the code.**

```
const Int Int::operator++(int) // postfix
{
    Int result{*this};  // copy of self
    ++m_val;                // change self
    return result;          // return copy
}
```

Note that the postfix version also returns an ***rvalue;*** a **const Integer**.

Symmetric Operators

The **symmetric operators** include the arithmetic and the relational operators. They are called the symmetric operators because **a+b** and **b+a** should have the same effect.

Assuming that **n** is an **Int** object, and that you can convert **int** to **Int**, you would like both of these addition operations to succeed:

```
Int n{35};
auto a = n + 10;    // Probably OK
auto b = 15 + n;    // Depends on how written
```

1. If the **Int** constructor is **explicit**, **neither** addition will succeed, since **10** is not an **Int** object.
2. If **operator+()** is a **member operator**, line 2 is: **n.operator+(10)**. The (non-**explicit**) **Int** constructor will convert **10** into an **Integer**, and the addition will succeed. Line 3 is: **15.operator(n)**, **which fails.**
3. If **operator+()** is written as a **non-member operator**, and the **Int** constructor is not **explicit**, then both **additions will succeed**.

For this reason, generally write such operators **as non-members**.

Because non-members have no access to the **private** data, this is only possible if the class has accessors, or, you have written **the equivalent side-effect operators.**

Addition

Here is a version of addition that uses **operator+=()**, defined earlier. Unlike the side-effect operator, addition (and the other arithmetic operators), should return a **const** object. (Arithmetic operators should return ***rvalues***, not ***lvalues***.)

```
const Int operator+(const Int& lhs, const Int& rhs)
{
    Integer result{lhs};
    result += rhs;      // already written
    return result;
}
```

1. Because this is a non-member, both operands are **explicit parameters**
2. Addition does not modify either operand, so both are **const**
3. The returned value is a temporary rvalue, so it is **const** as well
4. The algorithm creates a copy of the **lhs** parameter, modifies and returns it.

Input/Output Operators

You've already seen the input and output operators. Both **must be non-members**; the left-hand operand is the stream that you are writing to or reading from.

These operators don't have access to the **private** data members of the class. The **friend** keyword relaxes this restriction. Remember that the output operator is not a member operator; it's prototype just appears inside the class. Make sure that you don't qualify the operator with **Int::** when you define it:

```
class Int
{
    int m_val{0};   // private
public:
    friend
    std::ostream& operator<<(std::ostream&, const Int&);
};
```

Alternatively, if you don't wish to use **friend**, you can just add a **to_string()** helper **member** function, like the one shown here for **Int**.

```
string Int::to_string() const
{
    return to_string(m_val); // c++ 11 to_string
}
ostream& operator<<(ostream& out, const Int& i)
{
    out << i.to_string();
    return out;
}
```

The **to_string()** is a member function, but the **output operator** is a non-member. Output operators should do minimal formatting and **should not include newlines**.

Input Operators

Input operators are a little more complex, since you must deal with errors. Here is an example that handles the most common problems:

```
istream& operator>>(istream& in, Int& i)
{
    int n;              // temporary int
    in >> n;            // read new value
    if (in) i = Int{n}; // assign if succeed
    return in;          // return stream
}
```

1. The second parameter is an **Int&** (not a **const Int&**)
2. Create a temporary variable and read into it.
3. If successful, create a new **Int** and assign it to the output parameter.
4. Return the input stream (in a good or failed failed state).

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **make submit**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.