Lecture 17

Law of the Big Three and Virtual Constructors

- Our program now can do rather interesting things, e.g., to play a song given the correct sample builder.
- However, the classes we made are not very good classes: they give a lot of surprises to users. We'll try to make the class a bit nicer.

References:

- Third Edition: Section 11.7 Essential Operators.
- Third Edition: Section 11.3.4 Copying.
- Third Edition: Section 15.6.2 "Virtual Constructors".

Class weirdness

In general, when we create classes, we want to minimize surprises to users. Consider the following code:

```
void f(TimeLimiter tl) { /* Do something to tl */ }

SampleGenerator *build_generator() {
    TimeLimiter *t = new TimeLimiter(new SineNote(1000, 0.1), 0.5);
    f(*t);
    return t;
}
```

Will the program compile right?

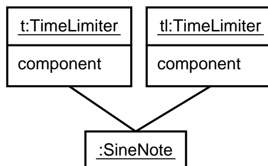
What will be returned by build_generator()?

What will happen afterwards?

Is it acceptable for TimeLimiter to has this behaviour?

What actually happened

- When we call f(*t), *t is passed as actual argument to f() as tl. Since tl is passed by value in f(), *t is copied to tl.
- This is done by copying all the fields of TimeLimiter from *t to tl, *including the component*.
- We have the following picture:



When f() exits, tl is destroyed. The destructor of TimeLimiter deletes the SineNote. Now *t→component* points to a deleted object.

The hidden side of C++

For each class X, four functions are automatically defined:

1. Default constructor: *X*::*X*()

   The default one does nothing.

2. Copy constructor: *X*::*X*(**const** *X*&).

   The default one copies another object by copying all fields.

3. Assignment operator: *X*& **operator**=(**const** *X*&).

   The default one copies another object by copying all fields, and return itself (i.e., ***this**).

4. Destructor: *X*::~*X*().

   The default one does nothing.

Where are they used?

Default constructor is used when you create an array of X. E.g.: X x[4] calls X::X() for each element.

Copy constructor is used when you:

- Construct an object in the form X x = y;
- Pass an argument by value.
- Return an argument by value.
- Catch an exception by value.

Destructor is called whenever an object is about to be destroyed.

All the default operators are assumed by STL.

Example

Where are these operators called?

```
class Complex {
public:
    double re, im;
}
Complex f(Complex a, Complex *b, Complex &c) {
    c = *b;
    return a;
}
void main() {
    Complex x;
    x.re = 2; x.im = 3;
    Complex y = x;
    Complex z[2];
    z[2] = f(x, &y, z[1]);
}
```

The problem

In a perfect world, nobody should attempt to copy a TimeLimiter. However, that doesn't mean that we can get away with the problem.

- People do make mistakes, especially ones that are made implicitly.
- The problem ends up in a random error that occur at run-time, making it very difficult to find.
- The user don't know whether the copy constructor do the right thing, unless he has the source and understands it.
- For many classes (e.g., Complex), the default operation is perfectly right. It is reasonable for users to expect a good copy constructor—especially if the semantics seems well defined.
- Many uses of a class (e.g., use in STL) requires good copy constructor and assignment operator. The lack of them severely limits the utilization of our classes.

Law of the Big Three

Our class is a special class: the default copy constructor (and the default assignment operator) doesn't do the right thing.

The set (destructor, copy constructor, assignment operator) is called the Big Three.

Law of the Big Three

> If you need non-default for any of the Big Three,
> then most likely you need non-default for all the Big Three.

Reason: Usually, destructors are non-default to release resources.

The default copy constructor and assignment operator make the same resources to be released twice.

The default assignment operator also makes some resources not released at all.

The simple solution: forbid copying and assignment

In some cases, a very simple solution suffices: forbid copying and assignment. Then any operation that implicitly perform copying causes an error.

To do so, define an empty private copy constructor and assignment operator to override the default ones:

```
class SampleGenerator {
public:
    SampleGenerator() {}
    virtual ~SampleGenerator() {}
    virtual bool advance(float t) = 0;
    virtual float get_sample() = 0;
private:
    SampleGenerator(const SampleGenerator&) {}// No copy!
    SampleGenerator& operator=(const SampleGenerator&) {
        return *this; }   // No assign!
};
```


The difficult solution: define copying and assignment

The easy solution is good if we really don't want copying and assignment. But what if we really need it?

Then we have to define our own copy constructors and assignment operators.

Note that SampleGenerator, SineNote, SquareNote and RandomNoise won't need a special destructor, and accordingly won't need any of the Big Three.

The copy constructor of TimeLimiter looks like this:

```
TimeLimiter::TimeLimiter(const TimeLimiter &time_limiter): t(0) {
    // Copy time_limiter.component to component
}
```

Note that we need pass by reference, otherwise we end up in infinite recursion.

Constructs virtually?

Now we have a problem: how to copy time_limiter.component?

Of course, the problem is that time_limiter.component is a SampleGenerator *, so **we don't know its type**. We cannot construct a copied SampleGenerator * without knowing its real type!

What we really want? Depending on the type of component, we want to get a different component. Sounds familiar?

We make a new virtual member function **clone()**, which does the actual copying. Since clone is not a constructor, it can be virtual.

On the other hand, **clone()** is defined by the individual derived classes, so they know what type to create and how to create it.

Now we get an object constructed via a virtual function: "**virtual constructor**"!

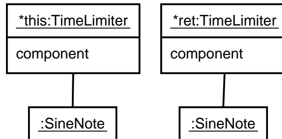
Easier clones

```
class SampleGenerator {
public:
    ...
    // Each sub-class must provide its own clone()
    virtual SampleGenerator *clone() = 0;
};

// SquareNote and RandomNoise are similar
class SineNote: public SampleGenerator {
public:
    SampleGenerator *clone() {
        SineNote * ret = new SineNote(freq, amp);
        ret->time = time;
        return ret;
    }
};
```

Clone for TimeLimiter

```
class TimeLimiter: public SampleGenerator {
public:
    ...
    SampleGenerator *clone() {
        TimeLimiter *ret =
            new TimeLimiter(component→clone(), length);
        ret→time = time;
        return ret;
    }
};
```



Code for assignment

Once we have clone, we can write our assignment operator for Time-Limiter:

```
class TimeLimiter: public SampleGenerator {
public:
    ...
    TimeLimiter(TimeLimiter &t) {
        component = t.component→clone();
        time = t.time;
        length = t.length;
    }
}
```

Note that this would work for any component that defines a correct *clone*() function.

Assignment operator

Whenever we perform something like x = y for two objects x and y, the assignment operator is performed.

- It is very similar to copy constructor in consequence: it should make x an exact copy of y.
- It is very different to copy constructor in principle: before the assignment, x is already there representing another object.
- It creates two more problems for us to solve:
  - We must **clean up** the object that is already there before the assignment.
  - We must handle a case in which the new object is just the same object (i.e., **self assignment**).
- Usually some code can be reused. But *don't make these code public!*

Assignment operator for TimeLimiter

As we have said, the big-three comes together. Default assignment operator of TimeLimiter won't work.

One attempt:

```
class TimeLimiter: public SampleGenerator {
public:
    ...
    TimeLimiter& operator=(const TimeLimiter& t) {
        delete component;
        component = t.component→clone();
        time = t.time; length = t.length;
        return *this;
    }
};
```

Problem: this cannot handle self-assignment: what will happen on "x=x"?

Revised code

One possible strategy is to do the copy before the delete:

```
class TimeLimiter: public SampleGenerator {
public:
    ...
    TimeLimiter& operator=(const TimeLimiter& t) {
        SampleGenerator *new_component = t.component→clone();
        delete component;
        component = new_component;
        time = t.time; length = t.length;
        return *this;
    }
};
```

Another strategy is to do explicit testing, by adding **if** (&*t*==**this**) **return**; at the beginning of the code. But this would make the code exception unsafe. (What will happen if *t.component*→*clone*() throws?)

Conclusion and Exercise

- Whenever the semantics is clear, classes should allow its objects to be copied.
- Law of the Big Three: the defaults for destructor, copy constructor and assignment operator either all works or all sucks.
- It is usually useful for these classes to provide a clone() member function, otherwise a recursive class will have difficulty to copy itself.

Exercise

- We (purposely) didn't touch on Composer. It is rather more difficult to do deal with Composer. Try to implement that. Read the CVS source code to see how it can be done.
- Try to add an operation reset() to allow generators to get back to starting state. (This will seriously affect the design of a Composer.) Write all non-default big-three's.