69. Object Members

Objectives

- Create a class with object members
- Use an implicit member constructor to instantiate objects
- Use an explicit member constructor to instantiate objects
- Learn the order in which member destructors are called
- Use delegation to shorten code

Object Members

No surprises: you can have members which are **objects**:

```
class ClubMember
{
private:
    int member_id_;
    Date signup_date_;
};

int main()
{
    ClubMember john; // john.signup_date_ is a
                     // Date object.

    return 0;
}
```

Here's another example:

```
class MyVehicle
{
private:
    char vin_[100];
    Vehicle vehicle_;
    Date purchase_date_;
};
```

Member Constructor: Implicit

Consider this example:

```
class C
{
public:
    C() { std::cout << "C()\n"; }
};

class D
{
private:
    C c;
};

int main()
{
    D d;

    return 0;
}
```

The class `D` contains a member `c` that is an object from class `C`. When creating `d`, the member `d.c` calls its default constructor `C::C()` automatically.

No constructor found

Now we'll look at a very common case that you may run into: the case when your compiler cannot find a constructor to call. For example,

```
class C
{
public:
    C(int a)
        : x_(a)
    {
        std::cout << "C()\n";
    }

private:
    int x_;
};

class D
{
private:
    C c_;
};
```

Error: no appropriate constructor. Why? Examine the above code carefully. Notice that the class `C` contains a constructor that takes an `int` parameter.

So what's the problem? Well, if an object of type `D` were created, then it would have to create its member, `c`. However, an `int` parameter for the constructor is necessary to create an object of type `C` (since `C` does not have a default constructor). The question is, assuming this code actually works, where would the `int` parameter come from? This produces an error because there is **no** `int` parameter passed into the constructor.

No surprise there. Recall that you must pass in all parameters for any function that takes them.

Member constructor: explicit

So what if I don't want to call the default constructor (the constructor that does not have any parameters), but some other constructor?? You can actually specify your constructor call ...

```
class C
{
public:
    C(int x)
        : x_(x)
    {
        std::cout << "C()\n";
    }

private:
    int x_;
};

class D
{
public:
    D(int x)
        : c_(a)
    {}

private:
    C c_;
};

int main()
{
    D d = D(0);
    return 0;
}
```

Member `c_` calls constructor `C::C(int)` explicitly.

Here's another example:

```
class ClubMember
{
public:
    ClubMember(int id, const Date & d)
        : member_id_(id), signup_date_(d)
    {}

private:
    int member_id_;
    Date signup_date_;
};
```

`signup_date` calls `Date::Date(const Date &)`, i.e., the copy constructor explicitly.

Exercise: Add a `get_signup_date()` method to the above class that does the obvious. Test your code with

```
int main()
{
    ClubMember john(42, Date(2015, 1, 1));
    john.get_signup_date().print();

    return 0;
}
```

Member destructor

Let's look at what happens with member destructors. Consider this example:

```
class C
{
public:
    C(int a): x_(a)
    {
        std::cout << "C()\n";
    }

    ~C()
    {
        std::cout << "~C()\n";
    }

private:
    int x_;
};

class D
{
public:
    D(int a)
        : c_(a)
    {}

    ~D()
    {
        std::cout << "~D()\n";
    }

private:
    C c_;
};

int main()
{
    C c = C(0);
    D d = D(1);

    return 0;
}
```

When object `d` does out of scope, destructor of `D`, i.e. `D::~D()`, is called. In `D::~D()`, member `c` calls its destructor `C::~C()`.

Constructor and destructor call order

If object `obj` has object members

```
obj1, obj2, obj3
```

declared in that order, then the constructor calls for the members are in this order:

```
obj1, obj2, obj3
```

When `obj` goes out of scope and the `obj` calls its destructor, then the order of destructor calls for members is:

```
obj3, obj2, obj1
```

Composition

Putting object(s) inside an object is called **composition**. That's one method of software re-use. There's also another type of software re-use in object-oriented programming called **inheritance**. You'll see that in many cases (but not all cases) , composition is better than inheritance.

Exercise. Complete the following circle class

```
// Circle.h

class Circle
{
public:
    Circle(int x0, int y0, int r0);

    void move(int dx, int dy)
    {
        x_ += dx; y_ += dy;
    }

    void print() const;

private:
    int x_, y_, r_;
};
```

```
// Circle.cpp

void Circle::print() const
{
    std::cout << "center: (" << x_ << ", " << y_
              << ")" << ", radius:" << r_ << '\n';
}

// Add other method definitions here
```

And test it with this program:

```
int main()
{
    Circle c(10, 10, 5);
    c.print();

    for (int i = 0; i < 10; i++)
    {
        c.move(2, 3);
        c.print();
    }

    return 0;
}
```

Now finish the following class:

```
class vec2i
{
public:
    // constructor
    // operator +=
    // operator<<

private:
    int x_, y_;
};
```

And rewrite your circle class:

```
class Circle
{
public:
    Circle(const vec2i & v, int r0);

    void move(const vec2i & v) { ... }

    void print(const vec2i & v) { ... }

private:
    vec2i center_;
    int r_;
};
```

And test it with this program:

```
int main()
{
    Circle c(vec2i(10, 10), 5);
    c.print();

    vec2i v(2, 3);

    for (int i = 0; i < 10; i++)
    {
        c.move(v);
        c.print();
    }

    return 0;
}
```

Delegation

The following is a common practice:

- Suppose every object of class `D` contains an object of class `C`.
- Suppose every object of class `C` has a method called `m()`.
- Then `D` can have a method call `m()` that calls the `m()` of class `C`.

For instance,

```
class C
{
public:
    void m()
    {
        std::cout << "C::m() ...\n";
    }
};

class D
{
public:
    void m()
    {
        std::cout << "D::m() ...\n";
        c.m();
    }
private:
    C c;
};
```

So when you call `d.m()`, `d` actually calls `c.m()` to do the work.

Here's another example:

Suppose you have a `vec2d` class (a vector class of 2 doubles). Suppose you have a `GameObject` class with a `vec2d` member called `pos` (the position vector in the screen). Suppose each `vec2d` object has a `get_x()` method. Then you might want to have a `get_x()` method in the `GameObject` class. This method calls the `p.get_x()` method.

```
GameObject alien;
std::cout << alien.get_x() << '\n';
```

Otherwise you would have to do this:

```
GameObject alien;
std::cout << alien.get_pos().get_x() << '\n';
```

In software, if you see something like

```
alien.get_pos().get_x()
```

with two "." operators (the "member operator"), then the delegation technique should probably be used. In well written software, there shouldn't be too many "." operators like the above. This is very subjective, but some recommend at most two ".". So the following is considered bad:

```
alien.get_rocket().load().fire()
```

Maybe have this instead:

```
alien.fire_rocket()
```

In fact some experts say there should be at most one "."

Exercise. Complete the following class

```
class TicTacToe
{
public:
    // constructor TicTacToe(int size):
    // initialize board to point size-by-size
    // chars of and set chars to ' '

    // copy constructor

    // destructor

    // set(int i, int j, char c): board[i][j] = c

    // get(int i, int j): return board[i][j]

    // operator<<

private:
    int size;
    char * board;
};
```

Test it with this:

```
int main()
{
    TicTacToe ttt(3);
    ttt.set(0, 0, 'X');
    ttt.set(1, 1, 'X');
    ttt.set(2, 2, 'O');

    std::cout << ttt << std::endl;

    return 0;
}
```

Now rewrite the `TicTacToe` class so that it uses a class that models pointers to arrays of `chars`:

```
class CharDynArr
{
public:
    // constructor CharDynArr(int size):
    //     initialize p to point to char[size]
    //     array is not initialized.

    // copy constructor

    // destructor

    // operator[] does the obvious thing

private:
    int size;
    char * p;
};
```

Now modify your `TicTacToe` class to use the above

```
class TicTacToe
{
public:
    ...
private:
    int size;
    CharDynArr board;
};
```