CS-1201 Object-Oriented Programming

`this` Pointer and Cascading Function Calls

Arbish Akram

Department of Computer Science
Government College University

this Pointer

- Each object has its own copy of data members.
- All objects share the same function definitions.
- The compiler uses the implicit `this` pointer to access the correct data members for each object.
- The `this` pointer itself is **not** part of the object: it is not counted when you take `sizeof` the object.

Example 1: Same Name as a Local Variable

When local variable names are the same as member variable names.

```
1  class Test {
2      private:
3          int x;
4      public:
5          void setX(int x) {
6              this->x = x; // Using 'this' to refer to the member variable
7          }
8          void print() {
9              cout << "x = " << x << endl;
10         }
11 };
12 int main() {
13     Test obj;
14     int x = 20;
15     obj.setX(x);
16     obj.print(); // Output: x = 20
17     return 0;
18 }
```

Why Do We Need this?

- A member function's **parameter** can have the **same name** as a **data member**, e.g. `setX(int x)` inside a class that also has a data member `x`.
- Inside the function body, the **local** `x` (the parameter) *hides* the **member** `x`; this is called *variable shadowing*.
- Plain `x = x;` would assign the parameter to **itself**; the member is never touched.
- In the call `obj.setX(x)` with `x = 20`: `this` points to `obj`, so `this->x` refers to `obj`'s **member** `x`.
- Plain `x` (no `this->`) refers to the local **parameter** `x`, value `20`.
- `this->x = x;` copies the parameter's value into the member: `obj.x` becomes `20`.

Example 2: Accessing an Object via a Pointer

```
1  class MyClass{
2   public:
3       int x;
4       void print() {
5           cout << "x = " << x << endl;
6       }
7   };
8
9   int main() {
10      MyClass obj;
11      obj.x = 42;
12      MyClass* ptr = &obj;
13      ptr->print(); // Accessing object using pointer
14      return 0;
15  }
```

Example 3: Implicit vs. Explicit this

```
1  class Test {
2      public:
3          Test(int = 0);
4          void print() const;
5      private:
6          int x;
7  };
8  // constructor
9  Test::Test(int value) : x(value) {}
10 void Test::print() const {
11     // implicitly use the this pointer to access the member x
12     cout << "x = " << x;
13     // explicitly use the this pointer and the arrow operator
14     cout << "\n  this->x = " << this->x;
15     // explicitly use the dereferences this pointer and the dot operator
16     cout << "\n(*this).x = " << (*this).x << endl;
17 }
18
19 int main( ) {
20     Test testObject(12);
21     testObject.print();
22     return 0;
23 }
```

Common Programming Error: `*this.x`

- The parentheses around `*this` are *required*: `(*this).x`.
- The dot operator `.` has **higher precedence** than `*`.
- Without parentheses, `*this.x` is read as `*( this.x )` — a compilation error, since the dot operator cannot be applied to a pointer.

Cascading Member Functions

- Cascading allows us to call multiple member functions for an object in a single statement.
- Let's say `sq` is an object of the class `Square`. To print the area and perimeter, we can call these methods like this:

```
sq.area();
sq.perimeter();
```

- Using cascading, we can achieve the same result in a single statement:

```
sq.area().perimeter();
```

Example: Cascading Function Calls

```
1  class Square
2  {
3    public:
4      int side;
5      Square area()
6      {
7        cout << "Area of the square is :" << side*side <<endl;
8        return *this;
9      }
10     Square perimeter()
11     {
12       cout << "Perimeter of the square is :" << 4*side <<endl;
13       return *this;
14     }
15 };
16 int main()
17 {
18   Square sq;
19   sq.side =3;
20   sq.area().perimeter(); //cascading function calls
21   return 0;
22 }
```

Both `area()` and `perimeter()` return `Square` **by value**, not `Square&`. Every link in the chain makes a fresh copy of the object.

Why Does Returning `*this` Enable Cascading?

- The dot operator `.` associates *left to right*.
- In `t.setHour(18).setMinute(30).setSecond(22);`:
  1. `t.setHour(18)` executes and returns a **reference** to `t`.
  2. The expression becomes `t.setMinute(30).setSecond(22);`
  3. `t.setMinute(30)` executes and returns a reference to `t`, leaving `t.setSecond(22);`
- For this to keep working **on the same object**, each function must return a *reference* (`Type&`), *not* a copy.
- Order matters: a function that returns `void` (e.g. a `print` function) **cannot** appear before another cascaded call.

Cascading with Reference Return Types

```
1  class Time {
2      public:
3          Time& setHour(int h) {
4              if (h >= 0 && h < 24) hour = h;
5              return *this;   // enables cascading
6          }
7          Time& setMinute(int m) {
8              if (m >= 0 && m < 60) minute = m;
9              return *this;   // enables cascading
10         }
11         void printStandard() const {
12             cout << hour << ":" << minute;
13         }
14     private:
15         int hour;
16         int minute;
17 };
18 int main() {
19     Time t;
20     // cascaded calls: each set function returns Time&
21     t.setHour(18).setMinute(30);
22     t.printStandard(); // 18:30
23     return 0;
24 }
```

Here `setHour` and `setMinute` return `Time&`, so the chain keeps operating on the **same** `t`, no copies are made, and any changes are visible through `t` afterward.