COMP 1633: Intro to CS II

More Classes

Charlotte Curtis

March 20, 2024

Where we left off

- Intro to object oriented programming
- Abstraction terminology
- Classes and objects - defining, creating, using

> *Textbook Sections 10.2-10.3*

```
class Cat {
public:
    void meow();
private:
    string name;
    int age;
};
```

Today's topics

- `const` correctness with classes
- Constructors and destructors
- Function and operator overloading

> *Textbook Sections 10.2, 11.2*

Our `Time` class

- Last lecture we defined a `Time` class, but it's a bit clunky to use:

```
Time now;
now.set(12, 30, 0);
now.write(cout);
```

- There's also no guarantee any of those functions won't modify the `Time`
- We can fix these things with **const correctness**, **constructors** (and destructors), and **operator overloading**

`const` correctness

How should I ensure that these functions don't modify the `Time` object?

```
void write(std::ostream &out);
int compare(Time other);
```

- `const` *before* a parameter means that the function will not modify it
- `const` *after* a **member function** means the function will not modify `this`
- As usual, if a function isn't going to modify something, `const` is a good idea

> *For that matter we might want to make `compare` take a `const Time &` instead of a `Time` - why?*

Remember `this`?

- `this` is a pointer to the **object** that the member function is being called on

```
void Time::five_o_clock_somewhere() {
    this->hour = 5; // usually don't explicitly use this->
}
```

```
Time now, later;
now.foo();
later.foo();
```

- No matter what class you're in, `this` is a `const` pointer of the class type
- In this example, you can imagine it being declared as `Time * const this`
- Adding the extra `const` means that `this` is a `const Time * const this`

`const` correctness

- As a general rule, if you *can* make something `const`, you *should*
- Caveat: A `const` member function can only call other `const` member functions (or use `const_cast`, but that's not a great idea)
- When you start using `const`, you should use it **consistently**

> *Let's go ensure `const` correctness in our `Time` class*

Constructors

- It's really useful to **initialize** variables when we declare them, but we can't use the `= {}` syntax with classes (in C++ 98)
- We can, however, define a **constructor**
- This is a special member function that is called when the object is created
- Syntax: **same name** as the class, **no return type**, and should be `public`

```
class Time {
public:
    Time();
};
```

Implementing a constructor

- Just like implementing a member function, but no return type:

```
Time::Time() {
    // or however you want the initial state
    hours = 0;
    minutes = 0;
    seconds = 0;
}
```

- If all you're doing is setting values, better to use an **initializer list**:

```
Time::Time() : hours(0), minutes(0), seconds(0) {}
```

Again, this is C++98 - no curly brace initialization

Using a constructor

- Constructors are called **implicitly** when the object is created:

```
Time now; // calls Time::Time()
Time *later = new Time; // also calls Time::Time()
```

- But it'd be useful to be able to set the time when we create the object:

```
Time now(3, 15, 2); // Can't call Time::Time(), too many arguments
```

- We can do this by **overloading** the constructor!

```
class Time {
public:
    Time(); // constructor with no arguments
    Time(int h, int m, int s); // constructor with 3 arguments
};
```

Side tangent: Function overloading

- A function is fully defined by its **signature** - its name and parameter types
- We can have multiple functions with the same name and different signatures!

```
int add(int a, int b);
double add(double a, double b);
...
int n = add(1, 2); // calls the first one
double x = add(1.5, 2.5); // calls the second one
```

- Our constructors might look something like:

```
Time::Time() : hours(0), minutes(0), seconds(0) {}
Time::Time(int h, int m, int s) : hours(h), minutes(m), seconds(s) {}
```

Side tangent: Function signatures

What counts as a different signature? Consider `void foo(int a, int b);`:

Function	Different?
`void foo(int a, int b, int c);`	Yes - number of parameters
`void foo(int a, char c);`	Yes - types of parameters
`void foo(char c, int a);`	Yes - order of parameters
`void foo(int c, int d);`	No - names of parameters don't matter
`void foo(const int a, int b);`	No - `const` doesn't matter
`bool foo(int a, int b);`	No - return type doesn't matter

Back to constructors

- All constructors must be named `ClassName` and have no return type
- Otherwise it's a standard function that can do anything:

```
Time::Time(bool now) {
    // query the system for the current time
}
```

- If you don't specify a constructor a **default** one is created that does nothing:

```
Time::Time() {}
```

- As soon as you specify a constructor, the default one goes away!

Destructors

- Every `new` needs a `delete`, so what if we dynamically allocate data in a class?
- We can define a **destructor** to be called when the object is destroyed
- Syntax is the same as a constructor, but with a `~` in front:

```
class Time {
public:
    ~Time();
};

Time::~Time() {
    // clean up any dynamically allocated data
}
```

- A destructor **cannot** take any parameters or have a return type

Using a destructor

- Destructors are called **implicitly** when the object is destroyed:

```
Time *later = new Time; // calls Time::Time()
delete later; // calls Time::~Time()
```

- Objects allocated on the **stack** are destroyed when they go out of scope:

```
void foo() {
    Time now; // calls Time::Time()
} // now goes out of scope, calls Time::~Time()
```

- Destructors are only needed if you have **dynamically allocated** data - our `Time` class actually doesn't need one

Better example: An `IntList` class

```
class IntList {
public:
    IntList();
    ~IntList();
    void append(int n);
    void write(std::ostream &out) const;
private:
    struct Node {
        int data;
        Node *next;
    };
    Node *head;
};
```

The `IntList` destructor

```
IntList::~IntList() {
    Node *curr = head;
    while (curr) {
        Node *temp = curr;
        curr = curr->next;
        delete temp;
    }
}
```

- Alternatively, this could be put in a **public member function** named `clear` and called from the destructor
- This would allow the client to clear the list without destroying the object

`const` and Constructors check-in 1/2

Which of the following is **not** a valid constructor declaration for the `Time` class?

A. `Time();`

B. `Time(int h, int m, int s);`

C. `Time(int h, int m);`

D. `Time(int h, int m, int s) const;`

E. `Time(std::string the_time);`

`const` and Constructors check-in 2/2

`const` **after** a member function declaration means:

A. The function can only be called on `const` objects

B. The `this` pointer is `const`, but the object it points to is not

C. The object the `this` pointer points to is `const`, but the pointer itself is not

D. The function will not modify the object it is called on

E. The function will not modify the arguments that are passed to it

Operator overloading

- Remember the `compare` function that I was too lazy to implement?

```
if (now.compare(later) == -1) {
    // do something
}
```

- I'd rather write this:

```
if (now < later) {
    // do something
}
```

- We can do this with **operator overloading**!

Operator overloading

- Operators are member functions with a bit of extra syntax:

```
class Time {
public:
    bool operator < (const Time &other) const;
};
```

- Now when we call `now < later`, the compiler sees:

```
now.operator < (later);
```

- The **calling object** (`now`) becomes the left hand side (LHS) of the operator, and the **argument** (`later`) becomes the right hand side (RHS)

Coming up next

- **Assignment 3** due Monday
- Lab: Class constructors and overloading
- **Assignment 4** will be refactoring assignment 3 with a `Leaderboard` as an abstract data type!