COMP 1633: Intro to CS II

Intro to Classes

Charlotte Curtis

March 18, 2024

Where we left off

- Various linked list algorithms:
  - Inserting a node
  - Searching for a value
  - Deleting a node
- Passing linked lists to functions
- Linked list variations

> *Textbook Chapter 13*

```
void clear_list(Node *&head) {
    while (head) {
        Node *temp = head;
        head = head->next;
        delete temp;
    }
}
```

Today's topics

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

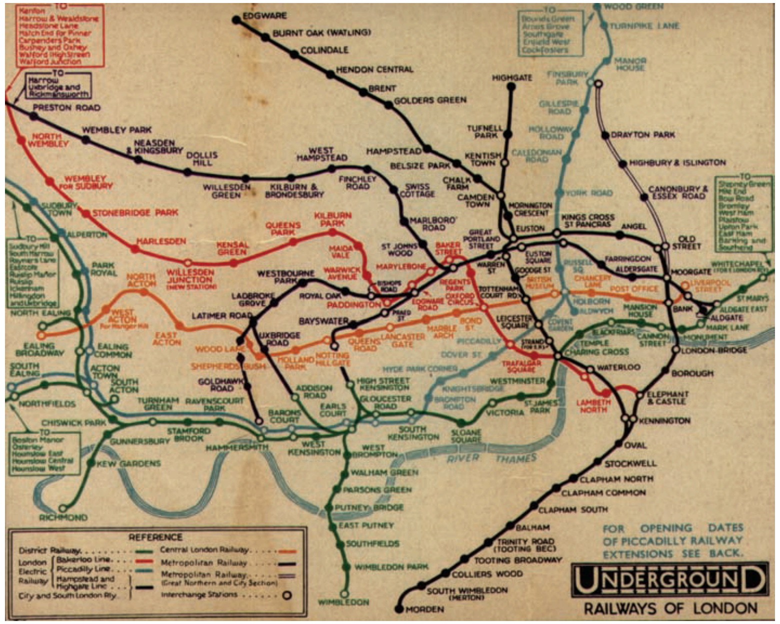
> *Textbook Sections 10.2-10.3*

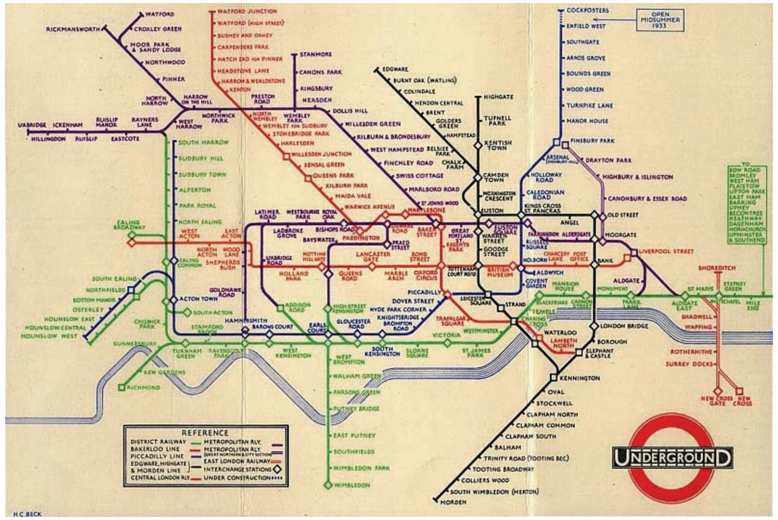
Object oriented programming

- So far we've been implementing solutions in a **procedural** style
- The **object oriented** approach is based on the idea that different **objects** can be interacted with in a different way
  - You can sit on a chair
  - You can draw with a pen
  - You can (probably) pick up a chair and a pen
  - Can you draw with a chair?
- In the OO approach, we can **encapsulate** data and functions in a `class` - an **abstract data type** that defines how an object can be interacted with

Abstraction

> *"The act of separating the essential qualities of an idea or object from the details of how it works or is composed" - Nell Dale and Chip Weems*





Abstraction in Computer Science

- A key concept that allows us to build complex systems by:
  - Understanding the overall system without understanding all the details
  - Focus on the parts of the system that are relevant to us
  - Use libraries and APIs without knowing how they've been implemented
- In general, two types of abstraction:
  - **Data abstraction** - hiding the details of how data is stored and accessed
  - **Procedural abstraction** - hiding the details of how a function is implemented

Procedural abstraction

Say I provide a header file and precompiled object file for the following functions:

```
// Reads a date formatted as year-month-day from source
void read_date(Date &date, std::istream &source);

// Writes the date to the output stream as year-month-day
void write_date(const Date &date, std::ostream &out);
```

- How are these functions implemented?
- Does it matter as long as they work?
- All the built-in functions we've been using are examples of abstraction!

Data abstraction

- Just as a function's behaviour can be separated from its implementation, **data abstraction** separates the properties of a **data type** from its implementation
- Essential for the design and planning of **custom data types**
- Every data type has two components:
    - **Domain** - the set of values that the type can take
    - **Operations** - things that can be done with the type
- An **abstract data type** (ADT) is a data type whose properties (domain and operations) are specified independent of the implementation

Example: `int`

- The domain of `int` is the set of all integers
- Operations:
  - Arithmetic: `+`, `-`, `*`, `/`, `%`
  - Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
  - Assignment: `=`
  - Increment/decrement: `++`, `--`
- How integers are actually implemented is irrelevant to us!

> *Note: According to our textbook, built-in types are ADTs*

Example: a new list type called `IntList`

- Domain, with some arbitrary decisions:
  - Homogenous linear collection of C++ `int`s
  - Minimum size 0, maximum size 100
  - Access by position starting from `1`
- Operations:
  - `insert`, `delete`, `retrieve` at a specific position
  - `search` for a value
  - `length`, `sort`, `print` the list

Example: a new list type called `IntList`

- If I handed you a `.h` and `.o` file implementing `IntList`, you could use it without knowing how it works
  - Internally, is it a linked list? An array?
  - Is the space allocated on the stack or the heap?
  - Is the length calculated on the fly, or stored in a variable?
- These are all details that you need to decide when **implementing** an ADT

> *To implement an ADT, we need to define a **class***

Classes

- A **class** is a blueprint for creating objects, much like how a `struct` is a blueprint for creating data structures

```
struct STime {
    int hours;
    int minutes;
    int seconds
};
```

```
class CTime {
    int hours;
    int minutes;
    int seconds;
    void write(std::ostream &out);
};
```

- A **class** is a **type** of object, just like `int` or `string` or `Node`
- **Member functions** are accessed using `.` or `->` just like member variables

In fact, in C++ a `struct` and a `class` are almost identical

Objects

- After defining a `class` (or a `struct`), we can create **objects** of that type
- Also called an **instance** of the class
- The syntax differs from a `struct` a little:

```
STime now = {5, 0, 0}; // struct
CTime bedtime; // class - can't use {} to initialize
bedtime.hours = 11; // uh oh, this doesn't work either!
```

- The main difference between a `struct` and a `class` is that the **member variables** (and functions) of a `class` are **private** by default
- **private** members can only be accessed by other members of the class

Class definition: general form

```
class ClassName {
public:
    // Public member functions (maybe some variables)
private:
    // Private member variables and functions
};
```

- The `public` and `private` keywords are **access modifiers**
- If you don't specify one or the other, `private` is assumed
- Good style to have `public` interface first, then `private` implementation details

Example: `Time` class

In general, anything **functions** that the user of the class needs to access should be `public`, and anything else should be `private` - including member variables!

`private` members

- `hours`
- `minutes`
- `seconds`

`public` members

- `write(std::ostream &out)`
- `set(int h, int m, int s)`
- `int compare(Time other)`
- `void increment()`

Side tangent: setters and getters

- Good practice to **encapsulate** member variables by making them `private`
- But this means we need a way to access them from outside the class
- **Getters** and **setters** are **public member functions** that allow us to access and modify **private member variables**
- This seems like extra work, but it allows us to do things like:
  - Check for valid values
  - Change the implementation of the class without affecting the user

> *All this being said, the C++ FAQ recommends avoiding trivial getters/setters*

`time.h`

Common for a class to have its own **header file** and **implementation file** (`.cpp`)

```
#ifndef TIME_H
#define TIME_H
class Time {
public:
    void write(std::ostream &out);
    void set(int h, int m, int s);
    int compare(Time other);
    void increment();
private:
    int hours;
    int minutes;
    int seconds;
};
#endif // TIME_H
```

Classes Check-in 1/2

How much memory is allocated when the following code is executed?

A. 0 bytes

B. 5 bytes

C. 8 bytes

D. 24 bytes

E. Undefined

```
class Student {
public:
    void set(int id, const char *name);
    void write(std::ostream &out);
private:
    char name[20];
    int id;
};
```

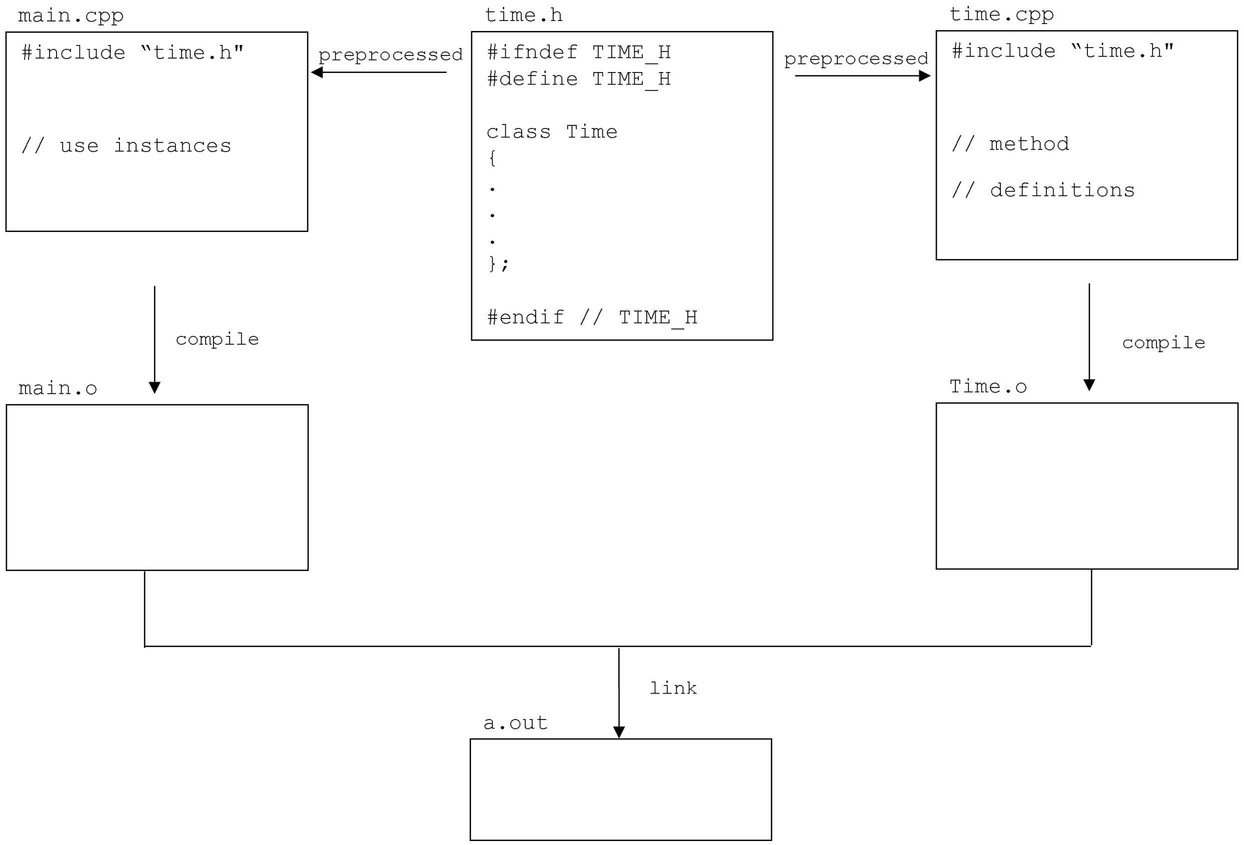
Classes Check-in 2/2

What is the main difference between a `struct` and a `class`?

A. A `struct` is a type of object, a `class` is a blueprint for creating objects
B. `struct`s can have public member variables, `class`es can't
C. `class`es can have functions, `struct`s can't
D. `struct` members are public by default, `class` members are private by default
E. `struct`s are allocated on the stack, `class`es are allocated on the heap

Using classes

- The program **using** the class is often called the **client**
- The client program `#include`s the header file to use the class
- Implementation is in the `.cpp` file - compiled separately



This is the same structure used in assignment 3!

Declaring objects

- Just like any other variable, we can declare objects of a class type, or pointers to objects of a class type

```
// In main.cpp
Time now; // object on the stack
Time *later = new Time; // pointer to object on the heap

now.set(3, 30, 0); // set the time for now
later->set(5, 0, 0); // set the time for later
```

- This isn't going to work just yet, we haven't actually **implemented** the class!
- All we've done is describe the class specification or **interface**

Implementing classes

- A `struct` just needs its declaration, but for a class we need to implement its member functions (aka **methods**)
- Syntax is a slight modification on the usual function definition:

```
// in time.cpp
ReturnType ClassName::func_name(Parameters) {
    // Function body
}
```

- `::` is called the **scope resolution operator**
- You've seen this already with `std::ostream`, `std::cout`, etc
- This allows multiple classes to have functions with the same name, like `set`

Implementing the `Time::set` function

```
// in time.cpp
void Time::set(int h, int m, int s) {
    hours = h;
    minutes = m;
    seconds = s;
}
```

- When implementing a member function, you don't need to use the `.` or `->` operators to access member variables
- If you do need to disambiguate, you can use the `this` pointer, which is **automatically created** as a pointer to the current **object**

  ```
  this->hours = h;
  ```

Calling member functions

Say we want to find the index position of the word "World" in a string:

Python

```python
hello = "Hello, World!"
pos = hello.find("World")
```

C++

```cpp
string hello = "Hello, World!";
int pos = hello.find("World");
```

- In both languages, `find` is a **member function** of the `string` class
- Calling a member function requests that **the object** perform some function
- In this case: "Hey, `hello`! Find the word 'World' and give me the index"

> *How is `find` implemented? No idea! Thanks, abstraction.*

Calling member functions of our own `class`

- Exactly the same:

```cpp
// in main.cpp
Time now;
now.set(3, 30, 0);
```

- The `now` object is automatically passed to the `set` function as the `this` pointer - it's not in the parameter list!
- If you have a pointer to an object, you can use the `->` operator:

```cpp
Time *later = new Time;
later->set(5, 0, 0);
```

- Just like `struct`s, this is equivalent to `(*later).set(5, 0, 0);`

Finishing off the class

- We have a few more functions to implement:
    - `write` - write the time to an output stream
    - `compare` - compare two times
    - `increment` - increment the time by one second
- The funkiest one is `compare` - in addition to the default `this` parameter, it needs another `Time` object

Coming up next

- Lecture: more on classes
- **Assignment 3** 🎉 - Due Monday, March 25
- Lab: Classes and objects

> *Textbook Sections 10.2-10.3*