User-Defined Data Types

CS 150 – C++ Programming I

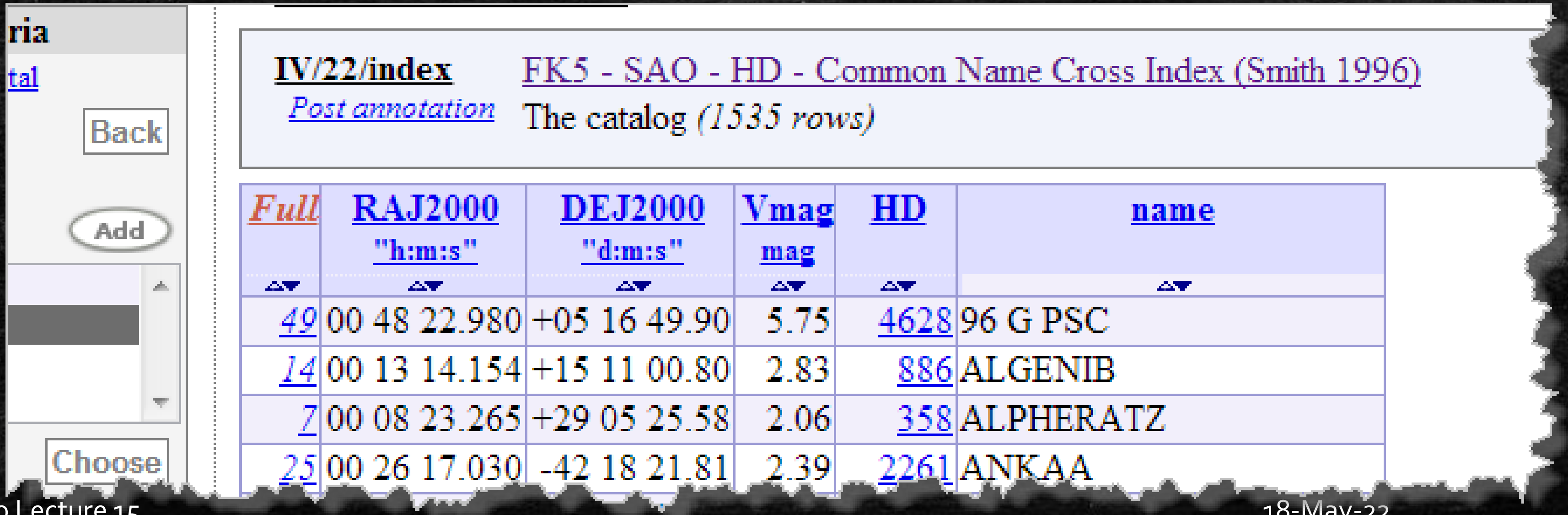
Lecture 15

Stream Review—Star Maps

- Exercise: read and process a star catalog `starcat.cpp`
  - Open and read the input file (see `stars.txt`)
  - Here are the specs for the file
    - x, y, z: - location
    - Draper number-a catalog identifier
    - Magnitude
    - Harvard Revised number—another identifier
    - Name (optional, may include secondary)
  - Print named stars: name, x, y, magnitude
- Exercise: finish *`starcat()`* and run tests

Heterogeneous Data Structures

- Each line in `stars.txt` consist of related information
  - Each portion contains information about a particular star
  - Simple variables aren't really flexible enough for such data
- We need a way to package up all of the parts into some kind of more complex, structured data of different types



IV/22/index FK5 - SAO - HD - Common Name Cross Index (Smith 1996)

Post annotation The catalog *(1535 rows)*

Full	RAJ2000 "h:m:s"	DEJ2000 "d:m:s"	Vmag mag	HD	name
49	00 48 22.980	+05 16 49.90	5.75	4628	96 G PSC
14	00 13 14.154	+15 11 00.80	2.83	886	ALGENIB
7	00 08 23.265	+29 05 25.58	2.06	358	ALPHERATZ
25	00 26 17.030	-42 18 21.81	2.39	2261	ANKAA

Records or Structures

- The generic CS term used for these are records
  - In C++, such records are called structures or `struct`
  - A user-defined collection of accessible heterogeneous data
- Here's the syntax for creating a structure definition

```
– struct Person
  {
      long long pID;         // named members
      std::string name;      // fully-qualified
      Date dob;              // other structure types
  };
```

Structure Tag

Don't Forget!!

The Structure Definition

- Structures are a new user-defined data type
  - Place the definition in a header file
  - It is an error if the definition is seen twice
  - Use header guards to prevent this
- Exercise: add your structure definition to `stars.h`
  -
    ```
    struct Star
    {
       double x, y, z, magnitude;
       int draper, harvard;
       string name1, name2;
    };
    ```

Structure Variables

- Use the structure definition to create structure variables
  - Like primitive types, such variables are uninitialized
  - `Star a, b; // two uninitialized stars`
- You may initialize `Star` variables in several ways
  - `Star d{}; // default initialize (all 0s)`

  -
    ```
    Star c = {.873, .032, .486, 2.07, 358, 15,
        "Rigel", "Beta" }; // aggregate initialize
    ```
  - `Star e(c);                // copy initialize`

Nested Structures

- Structures can contain others; given these . . .

  -
    ```
    struct Point3D { double x, y, z; };
    struct Names { string name1, name2; };
    struct Catalogs { int draper, harvard; };
    ```
- ...we can define the Star like this:

  -
    ```
    struct Star {
        Point3D location;
        double magnitude;
        Catalogs cats;
        Names names;
    };
    ```

Structure Access & Operations

- Directly access individual structure members using the member access operator (or dot) like this:
  - `cin >> a.name1 >> a.name2`
  - `cout << c.name1 << endl;`
- For a nested structure, just keep adding dots
  - `cout << s.location.x << endl;`
- You may also assign and copy entire structure variables
  - `a = b; // copies all members from b to a`
- Exercise: modify *starcat* to use your structure

Structures and Functions

- In the C language, structures are known as 2nd class types
  - Do not always act in the same way as the built-in types
  - `if (a == b) ...` Illegal if *a* and *b* are structured types
  - Fix by writing functions to supply the missing operations
- You can pass structure variables to functions
  - A function can also return a structure
  - Use the same rules for variable passing as for `string`
  - Pass by reference or const reference, never by value
  - `bool equal(const Star& a, const Star& b);`

Your Turn: Structure I/O

- Let's write some functions to print and read Star objects
  - ostream& print(ostream& out, const Star& s);
  - istream& read(istream& in, Star& s);
- Functions return the modified stream so it can be tested

  -
    ```
    ifstream in("stars.txt");
    Star s;
    while (read(in, s)) . . .
    ```
- Exercise: prototype & implement (stars.h & stars.cpp)
  - Uncomment first section of run() and make run

Overloaded Operators

- For any user-defined type you can overload most of the C++ operators to work with that type
  - Syntax for a binary operator (+, ==, >, etc)
    - `ret-type operator?(const Obj& lhs, const Obj& rhs)`
    - *lhs* means left-hand-side, *rhs* is right-hand-side
  - Replace *?* with the operator symbol
- Example: compare `Star` variables by magnitude

  -
    ```
    bool operator<(const Star& lhs, const Star& rhs) {
      return lhs.magnitude < rhs.magnitude;
    }
    ```

Overloaded I/O Operators

- Overloaded I/O operators look like this:
  - `ostream& operator<<(ostream& out,const Star& s)`
  - `istream& operator>>(istream& in, Star& s)`
- Almost same signatures as `print()` and `read()` functions, but with different names
  - You can use *`read()`* and *`print()`* to implement them
- Exercise: complete stars with `make test`

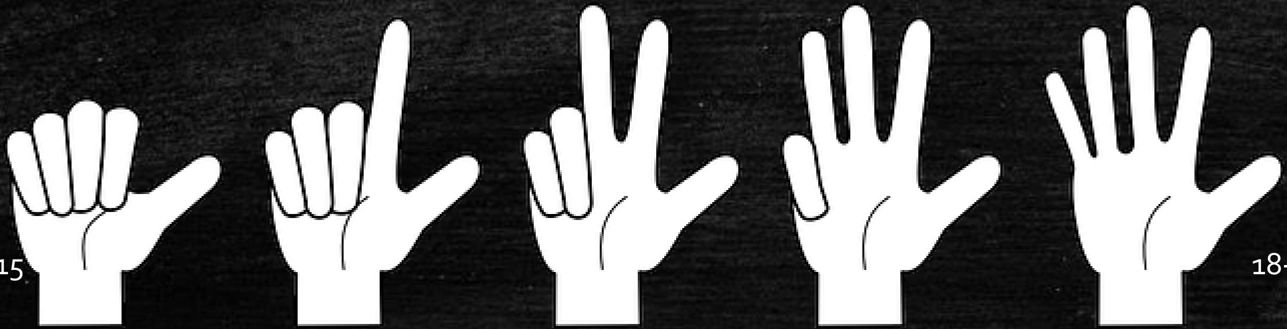
Enumerated Types

- User-defined scalar types are called enumerated types
  - Scalar meaning single value, vs. structured types
  - We can enumerate (list or count) each value
    - Example: the weekdays are Mon, Tue, Wed, Thu and Fri
- Can be written two ways:
  - `enum class Weekday { };        // newer (scoped)`
  - `enum Weekday { };              // older plain`
  - Scoped enumerations have less opportunity for errors

Defining an Enumerated Type

- A set of related named integer values which act like a type
  - You provide a name for each value
- Example: Suit values for a deck of cards (French deck)

  -
    ```
    enum class Suit {
        Clubs, Spades, Diamonds, Hearts
    };
    ```
  - Names separated by commas, with no ending semicolon
  - Use lowercase or proper case. Avoid UPPER_CASE



Using Scoped Enumerations

- Here are some of the things you can do
  - `Suit s;` // an enum variable
  - `s = Suit::Hearts;` // initializing
  - `s == Suit::Clubs;` // compare with != and ==
  - `switch(s) {` // use as a switch selector

    ```
      case Suit::Clubs: return "Clubs";
      case Suit::Hearts: return "Hearts";
      case Suit::Diamonds: return "Diamonds";
      case Suit::Spades: return "Spades";
      default: return "ERROR"; // or throw
    }
    ```

Why Use Enumerated Types?

- Consider a playing card structure with a suit and rank
  - What happens in each of these cases?

```
– struct Card {
    std::string rank, suit;
  };
– struct Card {
    int rank, suit;
  };
– struct Card {
   Rank rank;
   Suit suit;
  };
```

```
– Card a{"Ace", "Cubs"};
```

- Not caught by compiler

```
– const int Ace = 101:
  const int Clubs = 1001;
  Card b{Clubs, Ace};
```

- Not caught by compiler

```
– Card c{Rank::Ace, Rank::Clubs};
```

Using the Enumerated Type

- There is no built-in input/output with enumerated types
  - You may to `string` for output with a function

  -
    ```
    string to_string(Rank r) {
      switch (r) {
        case Rank::Ace: return "Ace";
        case Rank::Two: return "2";
        case Rank::Three: return "3";
    ..
    ```
- Exercise: complete `to_string()` for the `Coin` type

Card I/O Operators

-
  ```
  ostream& operator<<(ostream& out,const Card& c) {
        // Use out like you would cout
        return out;
  }
  ```
  - You'll need to use the `to_string()` for `Rank`, `Suit`

-
  ```
  istream& operator>>(istream& in, Card& s) {
        // Use in like you would cin
        return in;
  }
  ```
  - Use input in the form: `as`, `th`, `jc`

- Exercise - Write the Card I/O operators