Chapter 9: Technicalities: Classes, etc.

Plan for today

- We will talk about:
  - User-defined types
  - Classes and members
  - Interface and implementation
  - struct

Classes

- The **idea:**
  - A class directly represents a concept in a program
    - If you can think of “it” as a separate entity, it is likely that you should define a class to represent “it” in your program
    - Examples: vector, matrix, input stream, string, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - A class is a (user-defined) type that specifies how objects of its type are represented, how they can be created, used, and destroyed.
  - In C++ (as in most modern languages), a class is the key building block for large programs

Members and member access

- One way of looking at a class;

```
class X { // this class' name is X

    // data members
    // (they store information,
    //represent the current state)

   // function members
   // (they do things, using the information,
   // a set of operations that can be applied)
};
```

Members and member access

- Example:

```
class X {
public:
   int m;     // data member
   int mf(int v) { int old = m; m=v; return old; }
    // function member
};

X var;         // var is a variable of type X
var.m = 7;     // access var's data member m
int x = var.mf(9);  // call var's member function
```

Interface and Implementation

- We usually thing of class as having an *interface* plus an *implementation*
- The *interface* is the part of the class's declaration that its users access directly
  - identified by the label `public`
  - user's view of the class
- The *implementation* is that part of the class's declaration that its users access only indirectly through the interface
  - identified by the label `private`
  - implementer's view of the class

Interface and Implementation

- We usually thing of class as having an *interface* plus an *implementation*
- Example:

```
class X { // this class' name is X
public: // public members -- that's the interface to users
        // (accessible by all)
   // functions
   // types
   // data (often best kept private)
private:  // private members -- that's the implementation details
          // (accessible by members of this class only)
   // functions
   // types
   // data
};
```

Interface and Implementation

```
class Date { // this class' name is X
    int y, m, d; // class members are private by default
public:
    Date(int y, int m, int d);
    void addDay(int n); // increase the date by n days
    int month() { return m;}
    int day() { return d;}          int year() { return y;}
};
```

Interface and Implementation

```
class Date { // this class' name is X
   int y, m, d; // class members are private by default
public:
   Date(int y, int m, int d);
   void addDay(int n); // increase the date by n days
   int month() { return m;}
   int day() { return d;}          int year() { return y;}
};
```

- We can use it like this:

```
Date today(2023,3,2);          // OK
today.m = 4;                  // error: Date::m is private
cout << today.month() << endl; // OK
```

Interface and Implementation

```
class Date { // this class' name is X
    int y, m, d; // class members are private by default
public:
    Date(int y, int m, int d);
    void addDay(int n); // increase the date by n days
    int month() { return m;}
    int day() { return d;}          int year() { return y;}
};
```

- A date should be "valid". We try to design our types so that the values are guaranteed to be valid; we hide the representation, provide a constructor that creates only valid objects, and design all member functions to expect valid values and leave only valid values behind when they return.

Interface and Implementation

- The value of an object is often called its *state*, so
- The idea of a valid value is often refereed to as a *valid state* of an object

Interface and Implementation

- The value of an object is often called its *state*, so
- The idea of a valid value is often refereed to as a *valid state* of an object
- A rule for what constitutes a valid value is called an "*invariant*"
  - The invariant for Date ("a Date must represent a date in the past, present, or future") is unusually hard to state precisely
    - Remember February 28 (leap years), time zones, etc.

Interface and Implementation

- The value of an object is often called its *state*, so
- The idea of a valid value is often refereed to as a *valid state* of an object
- A rule for what constitutes a valid value is called an "*invariant*"
  - The invariant for Date ("a Date must represent a date in the past, present, or future") is unusually hard to state precisely
    - Remember February 28 (leap years), time zones, etc.
- If we can't think of a good invariant, we are probably dealing with plain data
  - If so, use a `struct`

Defining member functions and reporting errors

- Grab the file Date.h

(we will continue working on it)

Struct and Class

- There is a useful simplified notation for a class that has no private implementation details:
  - A `struct` is a `class` where members are public by default

```
struct X {
    int m;
    // …
};
```

- Means

```
class X {
public:
    int m;
    // …
};
```

  - `struct`ts are primarily used for data structures where the members can take any value

In-class work

- Design and implement `NamePairs` class hloding (name,age) pairs where name is a string and age is a double.
- Represent that as a `vector<string>` (called `name`) and a `vector<double>` (called `age`) members.
- Provide an input operator called `readNames()` that reads a series of names.
- Provide an input operator called `readAges()` that prompts the user for an age for each name.
- Provide a `print()` operation that prints out the `(name[i], age[i])` pairs (one per line) in the order determined by the `name` vector.

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html

- Class textbook