Chapter 9: Technicalities: Classes, etc.

Plan for today

- We will talk about:
  - Enumeration
  - `const` member functions
  - Operator overloading
  - Etc.

Date class

```
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
cout << today.month() << endl; // OK
```

Classes

- Why bother with the public/private distinction?
- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To ease debugging
    - Only a fixed set of functions can access the data
    - (known as the “round up the usual suspects” technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don’t really know who is using a public member

Date class constructor and isValid() member function

```
Date::Date(int y, int m, int d)
  :y{ yy }, m{ mm }, d{ dd }
{
  if (!isValid()) throw Invalid{};
   //check for validity
}
```

So far we only have a check for the month number.

Let’s add a vector of days in a month:

31 (january),28 (february),31,30,31,30,31,31,30,31,31

Enumerations

- An `enum` (enumeration) is a simple user-defined type, specifying its set of values (its enumerators) as symbolic constants
- For example:

```
enum class Month {
   jan = 1, feb, mar, apr, may, jun, jul, aug, sep,
oct, nov, dec
};
// the "body" of an enumeration is simply a list of its enumerators
Month m = Month::feb;   // but not   Month m = feb;
m = 7;              // error: can't assign int to Month
int n = m;        // error: we can't get the numeric value of a Month
```

~~Month mm = Month(7);   // convert int to Month (unchecked)~~

“Plain” Enumerations

- In addition to `enum class`es, also known as “scoped” enumerations, there are “plain” enumerations, that differ from scoped enumerations by implicitly “exporting” their enumerations to the scope of the enumeration and allowing implicit conversions to int
- For example:

```
enum ~~class~~ Month {
   jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
Month m = Month::feb;  or   Month m = feb;
m = 7;           // still an error: can’t assign int to Month
int n = m;     // ok, can assign a Month to an int
~~Month mm = Month(7);    // convert int to Month (unchecked)~~
```

Class interfaces

- Recall that class interface is the part of the class that its users access directly.
- What makes a good interface?
  - Keep it minimal, but complete
  - Provide essential operations:
    - constructors (default constructor or others)
    - Copy constructor and copy assignment (defaults to: copy members)
    - Destructor (defaults to: nothing, otherwise free all resources)
  - Use types to provide good argument checking
  - Identify nonmodifying memebr functions

Interfaces and “helper functions”

- Keep a class interface (the set of public functions) minimal
  - Simplifies understanding
  - Simplifies debugging
  - Simplifies maintenance

- When we keep the class interface simple and minimal, we need extra “helper functions” outside the class (non-member functions)
- E.g. == (equality) , != (inequality)
- next_weekday(), next_Sunday()

Operator overloading

- You can define almost all C++ operators for a class or enumeration operands
  - That’s often called “operator overloading”

```
enum class Month {
   jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
Month operator++(Month& m)  // prefix increment operator
{
   // “wrap around”:
   m = (m==Month::dec) ? Month::jan : Month(m+1);
   return m;
}
Month m = Month::nov;
++m; // m becomes dec
++m; // m becomes jan
```

Operator overloading

- You can define only existing operators
  - E.g., + - * / % [] () ^ ! & < <= > >=
- You can define operators only with their conventional number of operands
  - E.g., no unary <= (less than or equal) and no binary ! (not)
- An overloaded operator must have at least one user-defined type as operand
  - int operator+(int,int);   // error: you can't overload built-in +
  - Vector operator+(const Vector&, const Vector &);// ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - + should be addition, * be multiplication, [] be access, () be call, etc.
- Advice (not language rule):
  - Don't overload unless you really have to

Resources used for these slides

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