CSCI-1200 Data Structures — Spring 2014
Lecture 3 — Classes I & Order Notation

Review from Lecture 2

- Vectors are dynamically-sized arrays
- Vectors, strings and other containers should be:
  - passed by reference when they are to be changed, and
  - passed by constant reference when they aren't.

  If you forget the `&` and pass by value, the object will be copied which is expensive for containers with lots of elements. Note: This is unlike arrays, which are *not copied when passed by value.*
- Vectors can "contain" any type of objects, including strings and other vectors.

Today's Lecture

- Questions about Homework 1?
- Classes in C++
  - Types and defining new types
  - A `Date` class.
  - Class declaration: member variables and member functions
  - Using the class member functions
  - Class scope
  - Member function implementation
  - Classes vs. structs
  - Designing classes
- Introduction to Order Notation (Algorithm Analysis)

3.1 Types and Defining New Types

- What is a type? It is a structuring of memory plus a set of operations (functions) that can be applied to that structured memory.
- Examples: integers, doubles, strings, and vectors.
- In many cases, when we are using a class we don't know how that memory is structured. Instead, what we really think about is the set of operations (functions) that can be applied.
- To clarify, let's focus on strings and vectors. These are classes. We'll outline what we know about them:
  - The structure of memory within each class object
  - The set of operations defined
- We are now ready to start defining our own new types using classes.

3.2 Example: A Date Class

- Many programs require information about dates.
- Information stored about the date includes the month, the day and the year.
- Operations on the date include recording it, printing it, asking if two dates are equal, flipping over to the next day (incrementing), etc.

3.3 C++ Classes

- A C++ class consists of
  - a collection of member variables, usually `private`, and
  - a collection of member functions, usually `public`, which operate on these variables.
- `public` member functions can be accessed directly from outside the class,
- `private` member functions and member variables can only be accessed indirectly, through public member functions.
- We will look at the example of the `Date` class declaration.

3.4 Using C++ classes

- We have been using C++ classes (from the standard library) already this semester, so studying how the `Date` class is used is straightforward:

```
// Program:  date_main.cpp
// Purpose:  Demonstrate use of the Date class.

#include <iostream>
#include "date.h"

int main() {
  std::cout << "Please enter today's date.\n"
    << "Provide the  month, day and year: ";
  int month, day, year;
  std::cin >> month >> day >> year;
  Date today(month, day, year);

  Date tomorrow(today.getMonth(), today.getDay(), today.getYear());
  tomorrow.increment();

  std::cout << "Tomorow is ";
  tomorrow.print();
  std::cout << std::endl;

  Date Sallys_Birthday(9,29,1995);
  if (sameDay(tomorrow, Sallys_Birthday)) {
    std::cout << "Hey, tomorrow is Sally's birthday!\n";
  }

  std::cout << "The last day in this month is " << today.lastDayInMonth() << std::endl;
  return 0;
}
```

- **Important:** Each object we create of type `Date` has its own distinct member variables.
- Calling class member functions for class objects uses the "dot" notation. For example, `tomorrow.increment();`
- Note: We don't need to know the implementation details of the class member functions in order to understand this example. This is an important feature of object oriented programming and class design.

3.5 Exercise

Add code to `date_main.cpp` to read in another date, check if it is a leap-year, and check if it is equal to `tomorrow`. Output appropriate messages based on the results of the checks.

3.6 Class Declaration (`date.h`) & Implementation (`date.cpp`)

A class implementation usually consists of 2 files. First we'll look at the *header file* `date.h`

```
// File:     date.h
// Purpose:  Header file with declaration of the Date class, including
//   member functions and private member variables.

class Date {
public:
  Date();
  Date(int aMonth, int aDay, int aYear);

  // ACCESSORS
  int getDay() const;
  int getMonth() const;
  int getYear() const;

  // MODIFIERS
  void setDay(int aDay);
  void setMonth(int aMonth);
  void setYear(int aYear);
  void increment();

  // other member functions that operate on date objects
  bool isEqual(const Date& date2) const;  // same day, month, & year?
  bool isLeapYear() const;
  int lastDayInMonth() const;
  bool isLastDayInMonth() const;
  void print() const;                     // output as month/day/year

private:  // REPRESENTATION (member variables)
  int day;
  int month;
  int year;
};

// prototypes for other functions that operate on class objects are often
// included in the header file, but outside of the class declaration
bool sameDay(const Date &date1, const Date &date2); // same day & month?
```

And here is the other part of the class implementation, the *implementation file* `date.cpp`

```
// File:    date.cpp
// Purpose: Implementation file for the Date class.

#include <iostream>
#include "date.h"

using namespace std;

// array to figure out the number of days, it's used by the auxiliary function daysInMonth
const int DaysInMonth[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Date::Date() { //default constructor
  day = 1;
  month = 1;
  year = 1900;
}

Date::Date(int aMonth, int aDay, int aYear) { // construct from month, day, & year
  month = aMonth;
  day = aDay;
  year = aYear;
}
```

```
int Date::getDay() const {
  return day;
}

int Date::getMonth() const {
  return month;
}

int Date::getYear() const {
  return year;
}

void Date::setDay(int d) {
  day = d;
}

void Date::setMonth(int m) {
  month = m;
}

void Date::setYear(int y) {
  year = y;
}

void Date::increment() {
  if (!isLastDayInMonth()) {
    day++;
  } else {
    day = 1;
    if (month == 12) {  // December
      month = 1;
      year++;
    } else {
      month++;
    }
  }
}

bool Date::isEqual(const Date& date2) const {
  return day == date2.day && month == date2.month && year == date2.year;
}

bool Date::isLeapYear() const {
  return (year%4 ==0 && year % 100 != 0) || year%400 == 0;
}

int Date::lastDayInMonth() const {
  if (month == 2 && isLeapYear())
    return 29;
  else
    return DaysInMonth[ month ];
}

bool Date::isLastDayInMonth() const {
  return day == lastDayInMonth();   // uses member function
}

void Date::print() const {
  std::cout << month << "/" << day << "/" << year;
}

bool sameDay(const Date& date1, const Date& date2) {
  return date1.getDay() == date2.getDay() && date1.getMonth() == date2.getMonth();
}
```

3.7 Class scope notation

- `Date::` indicates that what follows is within the scope of the class.
- Within class scope, the member functions and member variables are accessible without the name of the object.

3.8 Constructors

These are special functions that initialize the values of the member variables. You have already used constructors for string and vector objects.

- The syntax of the call to the constructor mixes variable definitions and function calls. (See `date_main.cpp`)
- "Default constructors" have no arguments.
- Multiple constructors are allowed, just like multiple functions with the same name are allowed. The compiler determines which one to call based on the types of the arguments (just like any other function call).
- When a new object is created, *EXACTLY one constructor for the object is called.*

3.9 Member Functions

Member functions are like ordinary functions except:

- They can access and modify the object's member variables.
- They can call the other member functions without using an object name.
- Their syntax is slightly different because they are defined within class scope.

For the `Date` class:

- The `set` and `get` functions access and change a day, month or year.
- The `increment` member function uses another member function, `isLastDayInMonth`.
- `isEqual` accepts a second `Date` object and then accesses its values directly using the dot notation. Since we are inside class `Date` scope, this is allowed. The name of the second object, `date2`, is required to indicate that we are interested in its member variables.
- `lastDayInMonth` uses the const array defined at the start of the .cpp file.

More on member functions:

- When the member variables are *private*, the only means of accessing them and changing them from outside the class is through member functions.
- If member variables are made *public*, they can be accessed directly. This is usually considered bad style and should not be used in this course.
- Functions that are not members of the `Date` class must interact with `Date` objects through the class public members (a.k.a., the "public interface" declared for the class). One example is the function `sameDay` which accepts two `Date` objects and compares them by accessing their day and month values through their public member functions.

3.10 Header Files (.h) and Implementation Files (.cpp)

The code for the Date example is in three files:

- The *header file*, `date.h`, contains the class declaration.
- The *implementation file*, `date.cpp`, contains the member function definitions. Note that `date.h` is `#include`'ed.
- `date_main.cpp` contains the code outside the class. Again `date.h` again is `#include`'ed.
- The files `date.cpp` and `date_main.cpp` are compiled separately and then linked to form the executable program.
- Different organizations of the code are possible, but not preferable. In fact, we could have put all of the code from the 3 files into a single file `main.cpp`. In this case, we would not have to compile two separate files.
- In many large projects, programmers establish follow a convention with two files per class, one header file and one implementation file. This makes the code more manageable and is recommended in this course.

3.11 Constant member functions

Member functions that do not change the member variables should be declared `const`

- For example: `bool Date::isEqual(const Date &date2) const;`
- This must appear consistently in **both** the member function declaration in the class declaration (in the `.h` file) and in the member function definition (in the `.cpp` file).
- `const` objects (usually passed into a function as parameters) can ONLY use const member functions. *Remember, you should only pass objects by value under special circumstances. In general, pass all objects by reference so they aren't copied, and by const reference if you don't want/need them to change.*
- While you are learning, you will probably make mistakes in determining which member functions should or should not be const. Be prepared for compile warnings & errors, and read them carefully.

3.12 Exercise

Add a member function to the `Date` class to add a given number of days to the Date object. The number should be the only argument and it should be an unsigned int. Should this function be `const`?

3.13 Classes vs. structs

- Technically, a `struct` is a `class` where the default protection is `public`, not `private`.
  - As mentioned above, when a member variable is `public` it can be accessed and changed directly using the dot notation: `tomorrow.day = 52;` We can see immediately why this is dangerous (and an example of bad programming style) because a day of 52 is invalid!
- The usual practice of using `struct` is all public members and no member functions.

Rule for the duration of the Data Structures course: You may not declare new struct types, and class member variables should not be made public. This rule will ensure you get plenty of practice writing C++ classes with good programming style.

3.14 C++ vs. Java Classes

- In C++, classes have sections labeled `public` and `private`, but there can be multiple public and private sections. In Java, each individual item is tagged public or private.
- Class declarations and class definitions are separated in C++, whereas they are together in Java.
- In C++ there is a semi-colon at the very end of the class declaration (after the }).

3.15 Designing and implementing classes

This takes a lot of practice, but here are some ideas to start from:

- Begin by outlining what the class objects should be able to do. This gives a start on the member functions.
- Outline what data each object keeps track of, and what member variables are needed for this storage.
- Write a draft class declaration in a `.h` file.
- Write code that uses the member functions (e.g., the `main` function). Revise the class `.h` file as necessary.
- Write the class `.cpp` file that implements the member functions.

In general, don't be afraid of major rewrites if you find a class isn't working correctly or isn't as easy to use as you intended. This happens frequently in practice!

3.16 Exercise

What happens if the user inputs `2 30 2012` into the program? How would you modify the `Date` class to make sure *illegal dates* are not created?

3.17 Algorithm Analysis

Why should we bother?

- We want to do better than just implementing and testing every idea we have.
- We want to know why one algorithm is better than another.
- We want to know the best we can do. (This is often quite hard.)

How do we do it? There are several options, including:

- Don't do any analysis; just use the first algorithm you can think of that works.
- Implement and time algorithms to choose the best.
- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
- Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

3.18 Exercise: Counting Example

- Suppose `arr` is an array of `n` doubles. Here is a simple fragment of code to sum of the values in the array:

```
double sum = 0;
for (int i=0; i<n; ++i)
  sum += arr[i];
```

- What is the total number of operations performed in executing this fragment? Come up with a function describing the number of operations *in terms of n*.

3.19 Exercise: Which Algorithm is Best?

A venture capitalist is trying to decide which of 3 startup companies to invest in and has asked for your help. Here's the timing data for their prototype software on some different size test cases:

n	foo-a	foo-b	foo-c
10	10 u-sec	5 u-sec	1 u-sec
20	13 u-sec	10 u-sec	8 u-sec
30	15 u-sec	15 u-sec	27 u-sec
100	20 u-sec	50 u-sec	1000 u-sec
1000	?	?	?

Which company has the "best" algorithm?

3.20 Order Notation Definition

In this course we will focus on the intuition of order notation. This topic will be covered again, in more depth, in later computer science courses.

- Definition: Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- For example, algorithms requiring $3n+2$, $5n-3$, and $14+17n$ operations are all $O(n)$. This is because we can select values for k and n_0 such that the definition above holds. (What values?) Likewise, algorithms requiring $n^2/10+15n-3$ and $10000+35n^2$ are all $O(n^2)$.
- Intuitively, we determine the order by finding the *asymptotically dominant term (function of n)* and throwing out the leading constant. This term could involve logarithmic or exponential functions of n. Implications for analysis:
  - We don't need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.
- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

3.21 Common Orders of Magnitude

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- $O(\log n)$, *a.k.a. LOGARITHMIC.* e.g., dictionary lookup, binary search.
- $O(n)$, *a.k.a. LINEAR.* e.g., sum up a list.
- $O(n \log n)$, e.g., sorting.
- $O(n^2)$, $O(n^3)$, $O(n^k)$, *a.k.a. POLYNOMIAL.* e.g., find closest pair of points.
- $O(2^n)$, $O(k^n)$, *a.k.a. EXPONENTIAL.* e.g., Fibonacci, playing chess.

3.22 Exercise: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`. Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

```
int loc=0;
bool found = false;
while (!found && loc < n) {
  if (x == foo[loc]) found = true;
  else loc++;
}
if (found) cout << "It is there!\n";
```

3.23 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size array, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.
- The last is the most common. The first is rarely used.
- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

3.24 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the "size" of the problem. For arrays and other "container classes" this will generally be the number of values stored.
- Decide what to count. The order notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations. For example, in the previous exercise, we could count the number of comparisons.
- Do the count and use order notation to describe the result.

3.25 Exercises: Order Notation

For each version below, give an order notation estimate of the number of operations as a function of `n`:

1.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    ++count;
```

2.
```
int count=0;
for (int i=0; i<n; ++i)
  ++count;
for (int j=0; j<n; ++j)
  ++count;
```

3.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=i; j<n; ++j)
    ++count;
```