CSCI-1200 Data Structures — Fall 2024
Lecture 2 — C++ Classes, part 1

Announcements

- Hopefully you're all settled with your C++ development environment, and you have finished all of the checkpoints for Lab 1 & 2, and you either submitted your final version of Homework 1, or you at least earned the 1 day extension and will finish it this evening.
- *If the above is NOT true for you, please go to TA/mentor office hours today 4-6pm and ask for help to get caught up. This course moves fast and the TAs, mentors, and I are here to make sure you succeed. :)*
- Keep working on the Crash Course in C++ lessons. If you score at least 80% on each of Lessons 1-11 by Tuesday Sept 10th @ 11:59pm you will earn an additional late day that can be used on any future Data Structures Homework.
- If you didn't do the Syllabus and Collaboration Policy Quiz (I forgot to set it as a prereq for Homework 1), please do it ASAP. It will be a prereq for Homework 2.

Review from Lecture 1 & Lab 2

- Vectors are dynamically-sized arrays. You can ask them how big they are, and add data/make them bigger!
- Vectors, strings and other containers (that can hold an arbitrarily large amount of data) should be:
  - passed by reference when they are to be changed, and
  - passed by constant reference when they aren't meant to be changed.

  If you forget the `&` and pass by value, the object will be copied which is expensive for containers with lots of elements.
- Vectors can "contain" any type of objects, including strings and other vectors.

Today's Lecture

- Classes in C++; Types and defining new types; Examples: Date and `Name` classes;
- Class declaration: member variables and member functions; Using the class member functions;
- Member function implementation; Class scope; Classes vs. structs; Designing classes;
- Compilation and linking of classes; and Defining a custom sort for class instances.

2.1 Types and Defining New Types

- What is a type?
  - It is a structuring of memory plus a set of operations that can be applied to that structured memory.
  - Every C++ object has a type (e.g., integers, doubles, strings, and vectors, etc., including custom types).
  - The type tells us what the data means and what operations can be performed on the data.
- The basic ideas behind classes are **data abstraction** and **encapsulation**:
  - In many cases when we are using a class, we don't know (don't need to know) exactly how that memory is structured. Instead, what we really think about what operations are available.
  - Data abstraction hides details that don't matter from the end user and identifies details that do matter.
  - The user sees only the interface to the object: the collection of externally-visible data and operations.
  - Encapsulation is the packing of data and functions into a single component.
  - A well-designed and well-implemented class might be used by multiple programs, now and in the future!
- Information hiding
  - Users have access to interface, but not implementation.
  - No data item should be available any more than absolutely necessary.

- 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.

2.2 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.

2.3 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 from outside the class, through public member functions.
- We will look at the example of the `Date` class declaration.

2.4 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.

2.5 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.

2.6 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.

// These "include guards" prevent errors from "multiple includes"
#ifndef __date_h__
#define __date_h__

class Date {
public:
  // CONSTRUCTORS
  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?

#endif
```

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"

// 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();
}
```

2.7 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.

2.8 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.*

2.9 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.

2.10 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 and other functions that are closely related to the class. Note that `date.h` is `#include`'ed.
- `date_main.cpp` contains the code for a specific program that uses the `Date` class. Again `date.h` again is `#include`'ed.
- Different organizations of the code are possible... 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.
  - But putting everything in one file would not allow us to easily re-use the `Date` class in other programs.
- In large C++ software projects, we generally follow a convention with two files per class, one header file and one implementation file. This makes the code more manageable and is recommended/required in this course.
- The files `date.cpp` and `date_main.cpp` are compiled separately and then linked to form the executable program.

  First, we compile each source code `.cpp` file (which incorporate the `.h` file) and produce `date.o` and `date_main.o` *object files* (these are not human-readable):

  ```
  g++ -c -Wall -Wextra -g date.cpp
  g++ -c -Wall -Wextra -g date_main.cpp
  ```

  Then, we create the executable `date.out` by *linking* the object files:

  ```
  g++ -o date.out date.o date_main.o
  ```

  Alternatively, we can skip the intermediate step of making the object files and do both compilation and linking in one line. *This is what we will do for small programs in this course.*

  ```
  g++ -Wall -Wextra -g -o date.out date.cpp date_main.cpp
  ```

- Note that we don't have to directly or separately compile our header file. Why not? The `#include` command essentially copy-pastes the contents of the header file into the other file. Thus the code in the header file is compiled when the implementation file (or files) that use it are compiled.

2.11 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.

2.12 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`?

2.13 Another Simple Class Example: Alphabetizing Names, name_main.cpp

```
#include <algorithm>
#include <iostream>
#include <vector>
#include "name.h"

int main() {
  std::vector<Name> names;
  std::string first, last;
  std::cout <<"\nEnter a sequence of names (first and last) for alphabetization\n";
  while (std::cin >> first >> last) {
    names.push_back(Name(first, last));
  }
  std::sort(names.begin(), names.end());
  std::cout << "\nHere are the names, in alphabetical order.\n";
  for (int i = 0; i < names.size(); ++i) {
    std::cout << names[i] << "\n";
  }
  return 0;
}
```

2.14 name.h

```
// These "include guards" prevent errors from "multiple includes"
#ifndef __name_h_
#define __name_h_

#include <iostream>
#include <string>

class Name {
public:
  // CONSTRUCTOR (with default arguments)
  Name(const std::string& fst="", const std::string& lst="");
  // ACCESSORS
  // Providing a const reference to the string allows the string to be
  // examined and treated as an r-value without the cost of copying it.
  const std::string& first() const { return first_; }
  const std::string& last() const { return last_; }
  // MODIFIERS
  void set_first(const std::string & fst) { first_ = fst; }
  void set_last(const std::string& lst) { last_ = lst; }
private:
  // REPRESENTATION
  std::string first_, last_;
};

// operator< to allow sorting
bool operator< (const Name& left, const Name& right);

// operator<< to allow output
std::ostream& operator<< (std::ostream& ostr, const Name& n);

#endif
```

- Create an instance of the Name class:
  `Name person("Tom","Smith");`
- Attempt to permanently change Tom's first name:
  ```
  const std::string &a = person.first();
  a[1] = 'i';  // will fail to compile because a is const
  ```
- Make a copy of the returned `string`:
  ```
  std::string b = person.first();
  b[1] = 'i';  // edit does not affect the person object
  ```

2.15 name.cpp

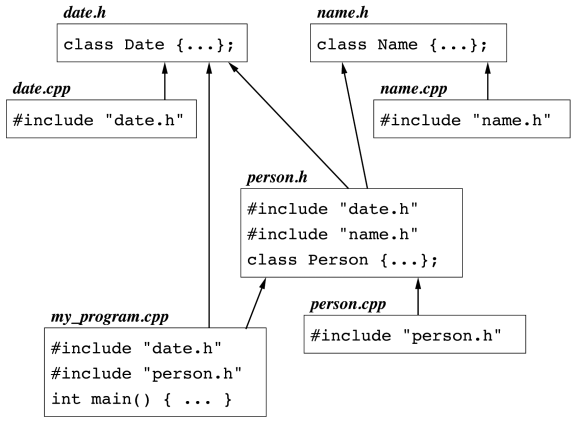
```
#include "name.h"

// Here we use member variable initializer list syntax to call the string
// class copy constructors (this way is technically more efficient).
Name::Name(const std::string& fst, const std::string& lst)
  : first_(fst), last_(lst)
{}
// Alternative implementation first calls default string constructor for the
// two variables, then performs an assignment in the body of the constructor
// function (this way is technically less efficient).  For this example,
// both versions will yield the same overall resulting program behavior.
/*
Name::Name(const std::string& fst, const std::string& lst) {
  first_ = fst;
  last_ = lst;
}
*/

// operator<
bool operator< (const Name& left, const Name& right) {
  return left.last()<right.last() ||
    (left.last()==right.last() && left.first()<right.first());
}
// The output stream operator takes two arguments: the stream (e.g., cout)
// and the object to print.  It returns a reference to the output stream
// to allow a chain of output.
std::ostream& operator<< (std::ostream& ostr, const Name& n) {
  ostr << n.first() << " " << n.last();
  return ostr;
}
```

2.16 Compilation of a Program with Multiple Classes

- Let's say we want to write a new program that wanted to use both of the classes we defined earlier, and also a third class named `Person` – that itself uses both `Name` and `Date` classes.
- Our collection of files might look like:



- Then to compile this program we would run:

  ```
  g++ -Wall -Wextra -g -o my_program.out date.cpp name.cpp person.cpp my_program.cpp
  ```

  Note that we never put the `.h` *files in the compilation command.*

- Alternatively, if these 7 files are the only code files in the current directory, we can use the '`*`' wildcard character which will match the 4 files files ending in `.cpp` in the current directory:

  ```
  g++ -Wall -Wextra -g -o my_program.out *.cpp
  ```

2.17 Classes vs. structs

- The C language doesn't have classes, but it does have `structs`. So we also have `structs` in C++.
- Technically, a `struct` is a `class` where the default protection is `public`, not `private`.
  - As mentioned above, if 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 would be 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 member variables and no member functions.

Rule for the duration of the Data Structures course: Write classes, not structs. Your class member variables should *not* be made public. This rule will ensure you get plenty of practice writing C++ classes with good programming style.

2.18 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 }).

2.19 C++ vs. Python Classes

- Python classes have a single constructor, __init__.
- Python is dynamically typed. Class attributes such as members are defined by assignment.
- Python classes do not have private members. Privacy is enforced by convention.
- Python methods have an explicit *self* reference variable.

2.20 Designing and implementing classes

Good software design requires 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!

2.21 Exercise

What happens if the user inputs `2 30 2019` into the program? How would you modify the `Date` class to make sure *illegal dates* are not created?

2.22 Providing Comparison Functions to Sort

- If we make an STL `vector` of `Date` objects, can we sort them? Yes! How?

```
std::vector<Date> dates;
dates.push_back(tomorrow);
dates.push_back(Sallys_Birthday);
dates.push_back(Date(10,26,1995));
```

- If we used:

```
sort(dates.begin(), dates.end());
```

  the sort function would try to use the `<` operator on `Date` objects to sort the dates, just as it uses the `<` operator on `int`s or `float`s. However, this doesn't work because there is no such operator on `Date` objects.

- Fortunately, the sort function can be called with a third argument, a comparison function. E.g.,:

```
sort(dates.begin(), dates.end(), earlier_date);
```

  Where `earlier_date` is a helper function we define in `date.h` and `date.cpp` that takes two const references to `Date` objects and returns true if and only if the first argument should be considered "less" than the second in the sorted order.

```
bool earlier_date (const Date& a, const Date& b) {
  if (a.getYear() < b.getYear() ||
      (a.getYear() == b.getYear() && a.getMonth() < b.getMonth()) ||
      (a.getYear() == b.getYear() && a.getMonth() == b.getMonth() && a.getDay() < b.getDay()))
    return true;
  else
    return false;
}
```

- That's great! But wait, how does `sort` work with STL `string`s?

2.23 Operator Overloading

- A second option for sorting is to define a function that creates a `<` operator for `Date` objects! At first, this seems a bit weird, but it is extremely useful.
- Let's start with syntax. The expressions `a < b` and `x + y` are really function calls!
  Syntactically, they are equivalent to `operator< (a, b)` and `operator+ (x, y)` respectively.
- When we want to write our own operators, we write them as functions with these weird names.
- For example, if we write:

```
bool operator< (const Date& a, const Date& b) {
  return (a.getYear() < b.getYear() ||
          (a.getYear() == b.getYear() && a.getMonth() < b.getMonth()) ||
          (a.getYear() == b.getYear() && a.getMonth() == b.getMonth() && a.getDay() < b.getDay()));
}
```

  then the statement `sort(dates.begin(), dates.end());`
  will sort `Date` objects into chronological order.

- Really, the only weird thing about operators is their syntax.
- We will have many opportunities to write operators throughout this course. Sometimes these will be made class member functions, but more on this in a later lecture.

2.24 A Word of Caution about Operators

- Operators should only be defined if their meaning is intuitively clear.
- Sorting `Date` objects makes sense because arguably chronological is the only natural, universally agreed-upon way to do this.
- Similarly, sorting STL `string` objects makes sense because alphabetical is the accepted order. So yes, the STL `string` class has overloaded `operator<`, and that's why sorting them works like magic.
- In contrast, if we defined a `Person` class (storing their name, birthday, address, social security number, favorite color, etc.), we probably wouldn't agree on how to sort a vector of people. Should it be by name? Or by age? Or by height? Or by income?

  Instead, it would be better to have comparison helper functions that can be used as needed.
  E.g., `alpha_by_name`, `youngest_first`, `tallest_first`, etc.