Midterm Prep

Content:

- Chapter 5 (5-11) pg. 269 - 288 (19 pgs)
- Chapter 6 (6-10, menu-driven logic) pg. 324 - 326 (2 pgs)
- Chapter 7 pg. 381 - 448 (67 pgs)
- Chapter 9 pg. 503 - 549 (46 pgs)
- Chapter 12 pg. 665 - 709 (44 pgs)
- Chapter 13 pg. 719 - 802 (83 pgs)

Schedule:

✓ Su	Read Ch. 5, 6
✓ M	Read Ch. 7 | PC: 7x1
✓ T	Read Ch. 9 | Read Ch. 12 (part)
✓ W	———
✓ R	Read Ch. 12 (remainder) | Read Ch. 13
⊙ F	Compile: notes, templates, slides, quizzes, paper + pen
· Sa	Morning review ↳ trial online compilers

* Read includes slides

Day of:

one doc?

- ↳ compile quizzes
- ↳ print quizzes, put on tablet
- ↳ practice UML
- ↳ compile notes → templates for files
- ↳ print notes, put on tablet
- ↳ grab pen, pencils, printer paper
- ↳ syntax review
  - ↳ test from memory, especially ch. 12

Chapter 5 (5-11)

- data saved to files sustains
- data stored in vars and control properties stored temporarily in RAM
- Three-step file process
  - ↳ open file
  - ↳ process file
  - ↳ close file
- two file types
  - ↳ text: ASCII or Unicode
  - ↳ binary
- sequential vs direct access
  - ↳ random-access file / direct access file
- file stream object associated w/ specific file
- two ways to open files
  - ↳ .open()
  - ↳ ifstream input_file("...txt");
- closing files
  - ↳ temp data stored in file buffer. Closing file saves data in buffer to file.
  - ↳ some OS's limit number of open files
  - ↳ .close()
- read position
  - ↳ initialized to first byte of file
- >> stream operator automatically converts text encoded numeric data to numeric data type.
  - ↳ also works as boolean to detect if data read or not
    - ↳ while(input_file >> number)
- detect successful file open
  - ↳ if(input_file) if successfully opened
  - ↳ if(input_file.fail()) if failed to opened
- string literals are stored in memory as null terminated strings, not string objects
  - ↳ .c_str()

Chapter 7

- data types and storage reqs
  - ↳ bool (1 byte)
  - ↳ char (1 byte)
  - ↳ short (2 bytes)
  - ↳ int (4 bytes) (assumes: signed int)
    - ↳ unsigned (4 bytes)
  - ↳ long (4 bytes)
  - ↳ long long (8 bytes)
  - ↳ float (4 bytes)
  - ↳ double (8 bytes)
  - ↳ long double (8 - 16 bytes)
  - ↳ string (variable)

built-in primitive data types

- array size declarator
  - ↳ must be constant or literal
- subscript numbers can be stored as variables

- if array defined globally, all elements initialized to zero
  - ↳ locally, no default init. value
    - ↳ instead, will contain garbage data like all other local vars!

- initialization list
- static_cast <datatype> var
- partial initialization, set rest to 0
  - ↳ can't initialize elements randomly, must be in sequence from beginning
- while (count < ARRAY_SIZE && inputFile >> numbers [count])
- range-based for loop
  - ↳ range variable
  - ↳ for (dataType rangeVariable : array)
    statement;
    - ↳ datatype must match data type of array elements
- references vs. pointers
  - ↳ &val
  - ↳ val (actual value)
  - ↳ val (after declaring *val = nullPtr
  - ↳ val * (dereference from address)
- increments
  - ↳ ++score[2] pre-increment
  - ↳ score[2]++ post-increment
- void showValues (int nums[], int size)
  - ↳ pass array via fxn
- vector <int> array (10, 2)
  - ↳ declare size of array
  - ↳ initialized to value of 2
- vector <int> numbers {10, 20, 30, 40}

- Vector method functions
  - ↳ .at(element)
  - ↳ .push_back()
  - ↳ .pop_back()
  - ↳ .reverse()
  - ↳ .swap()
  - ↳ .size()
  - ↳ .resize()
  - ↳ .clear()
  - ↳ .empty()

Chapter 12

- .get(), .put() → optional delimiter
- getline(file, str, '\n')
- Program type: Filter
- .write()
  - ↳ fileObject.write(address, size)
    (&letter, sizeof(letter))
- .read()
  - ↳ fileObject.read(address, size)
    (&letter, sizeof(letter))
- reinterpret_cast<datatype>(value)

```
int x = 65;
char *ptr = nullptr;
```

  - ptr=reinterpret_cast<char*>(&x)
  - fileObject.write(reinterpret_cast<char*>(&x), sizeof(x))
- for structures, always open in ios::binary mode due to mix of data structure types
- sequential vs. random file access
- .seekp(), .seekg()
  - ↳ ios::beg ↳ must use file.clear() else eof flag will prevent functionality
  - ↳ ios::end
  - ↳ ios::cur
  - ↳ file.seekp(20L, ios::beg)
  - ↳ file.seekp(-20L, ios::end)
- .tellg(), .tellp()
  - ↳ helps w/ determining number of bytes in file

ios::app	
ios::ate	
ios::binary	
ios::in	
ios::out	
ios::trunc	
ios::eofbit	.eof()
ios::failbit	.fail()
ios::hardfail	
ios::badbit	.bad()
ios::goodbit	.good()
	.clear()

Chapter 13

- objects
  - ↳ data + functions
    - attributes + member fxns
    - (data) (procedures)
- encapsulation
- data hiding
- object reusability
- instance of class, instantiation of a class
- access specifiers
- Member functions
  - ↳ void getWidth() const;
- :: → scope resolution operator
- accessor + mutator
  - (getter) (setter)
- object's state
  - ↳ data in object at a given moment
- stale data
- → operator with pointers!

v1

```
class Rectangle...
Rectangle rect_1
Rectangle *rectptr = nullptr
rectptr = new Rectangle
rectptr = &rect_1
rectptr -> setLength(12)
rectptr -> setWidth(7)
delete rectptr
rectptr = nullptr
   or rectPtr
```

v2

```
#include <memory>
unique_ptr<Rectangle> rect_ptr(new Rectangle)
Rectangle rect_2
rect_ptr = &rect_2
rect_ptr -> set_length(8.3)
rect_ptr -> set_width(12.6)
rect_ptr = nullptr
```

→ alt way:

```
auto rect_ptr = make_unique<Rectangle>()
```

- class specification file
  - ↳ contains class declaration
- class implementation file
  - ↳ stores member function defs

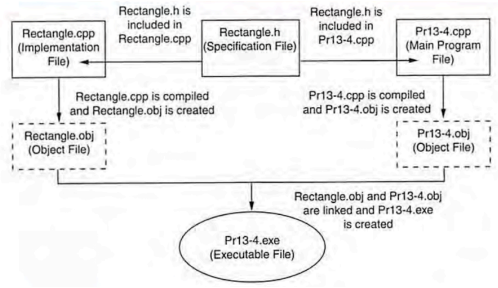
- #ifndef is an include guard
  - ↳ if not defined
- #define (defines macro)
- #endif

} pre-processor directives

Rectangle Example:

- Rectangle.h
- Rectangle.cpp
- main.cpp

create object files



- inline function: double getArea() const { return width * length }
- inline expansion
  - ↳ improved performance (generally)

Contents of Rectangle.h (Version 1)

```
1  // Specification file for the Rectangle class.
2  #ifndef RECTANGLE_H
3  #define RECTANGLE_H
4
5  // Rectangle class declaration.
6
7  class Rectangle
8  {
9     private:
10       double width;
11       double length;
12    public:
13       void setWidth(double);
14       void setLength(double);
15       double getWidth() const;
16       double getLength() const;
17       double getArea() const;
18 };
19
20 #endif
```

- stack
- constructors not executed by explicit fxn calls
  - ↳ initializes object's attributes
- member initialization lists

```
Rectangle()          vs.   Rectangle(): width(0.0), length(0.0) { }
{ width = 0.0
  length = 0.0 }
```

initialization take place before body of constructor executes

- inline initialization
- default constructor
  - ↳ Rectangle *rect_ptr = nullptr (constructor not executed)
    rect_ptr = new Rectangle (constructor executes)
- as long as constructor can be called w/ no arguments, it's the default constructor (can have parameters w/ default arguments)
- destructors

Aside:

```
private:
    char *name;    // The name
    char *phone;   // The phone number
public:
    // Constructor
    ContactInfo(char *n, char *p)
    { // Allocate just enough memory for the name and phone number.
        name = new char[strlen(n) + 1];
        phone = new char[strlen(p) + 1];

        // Copy the name and phone number to the allocated memory.
        strcpy(name, n);
        strcpy(phone, p); }
```

↳ common approach to dynamically allocating memory for a C-string. strlen() only tallies up chars BEFORE the null terminator '\0'. Must add 1 to account for it.

↳ just use strings!

- Dynamically create new class object
  - ↳ ContactInfo *ptr = nullptr
    ptr = new ContactInfo("Armand", "123-4567")
- constructor delegation

```
class Contact {
    string name, email, phone;
public:
    Contact() : Contact("", "", "") { }   // delegate to main constructor

    Contact(string n, string e, string p)
        : name(n), email(e), phone(p) { }
};
```

Contents of Rectangle.cpp (Version 1)

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

//***********************************************************
// setWidth sets the value of the member variable width.    *
//***********************************************************

void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
    else
    {
        cout << "Invalid width\n";
        exit(EXIT_FAILURE);
    }
}

//***********************************************************
// setLength sets the value of the member variable length.  *
//***********************************************************

void Rectangle::setLength(double len)
{
    if (len >= 0)
        length = len;
    else
    {
        cout << "Invalid length\n";
        exit(EXIT_FAILURE);
    }
}

//***********************************************************
// getWidth returns the value in the member variable width. *
//***********************************************************

double Rectangle::getWidth() const
{
    return width;
}

//***********************************************************
// getLength returns the value in the member variable length. *
//***********************************************************

double Rectangle::getLength() const
{
    return length;
}

//***********************************************************
// getArea returns the product of width times length.       *
//***********************************************************

double Rectangle::getArea() const
{
    return width * length;
}
```

Program 13-5

```
// This program uses the Rectangle class, which is declared in
// the Rectangle.h file. The member Rectangle class's member
// functions are defined in the Rectangle.cpp file. This program
// should be compiled with those files in a project.
#include <iostream>
#include "Rectangle.h" // Needed for Rectangle class
using namespace std;

int main()
{
    Rectangle box;     // Define an instance of the Rectangle class
    double rectWidth;  // Local variable for width
    double rectLength; // Local variable for length

    // Get the rectangle's width and length from the user.
    cout << "This program will calculate the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;
```

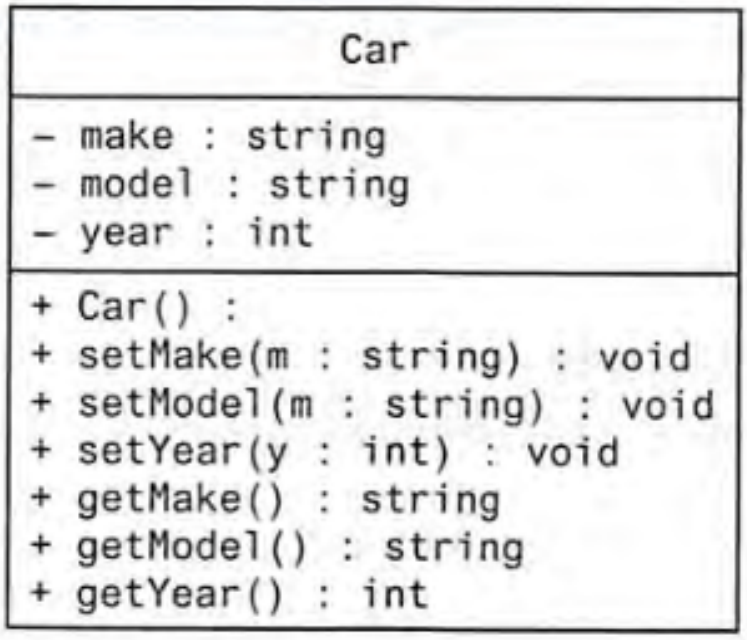
(program continues)

Program 13-5 *(continued)*

```
    // Store the width and length of the rectangle
    // in the box object.
    box.setWidth(rectWidth);
    box.setLength(rectLength);

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

UML Diagrams



STL packages, methods, functions:

Ch. 12

<fstream> : ifstream, ofstream, fstream

↳ flags : ios:: in, out, ate, app, trunc, binary

↳ ios:: in | ios:: out preserves contents

Error flags/fxns → recoverable → unrecoverable

↳ ::eofbit, failbit, hardfail, badbit, goodbit

↳ .eof(), .fail(), .bad(), .good(), .clear()

Fxns, methods

↳ >>, <<

↳ get(), put()

↳ in.get(ch)
or ch = in.get()

↳ obj = getline(dataFile, str, '\n')

↳ seekg(), seekp()

↳ ios:: beg, end, cur)

↳ seekg(-10L, ios:: end)

↳ tellg(), tellp()

↳ .peek() reads next char w/o moving

Binary Files

↳ .write(&letter, sizeof(data))

↳ .read(&letter, sizeof(data))

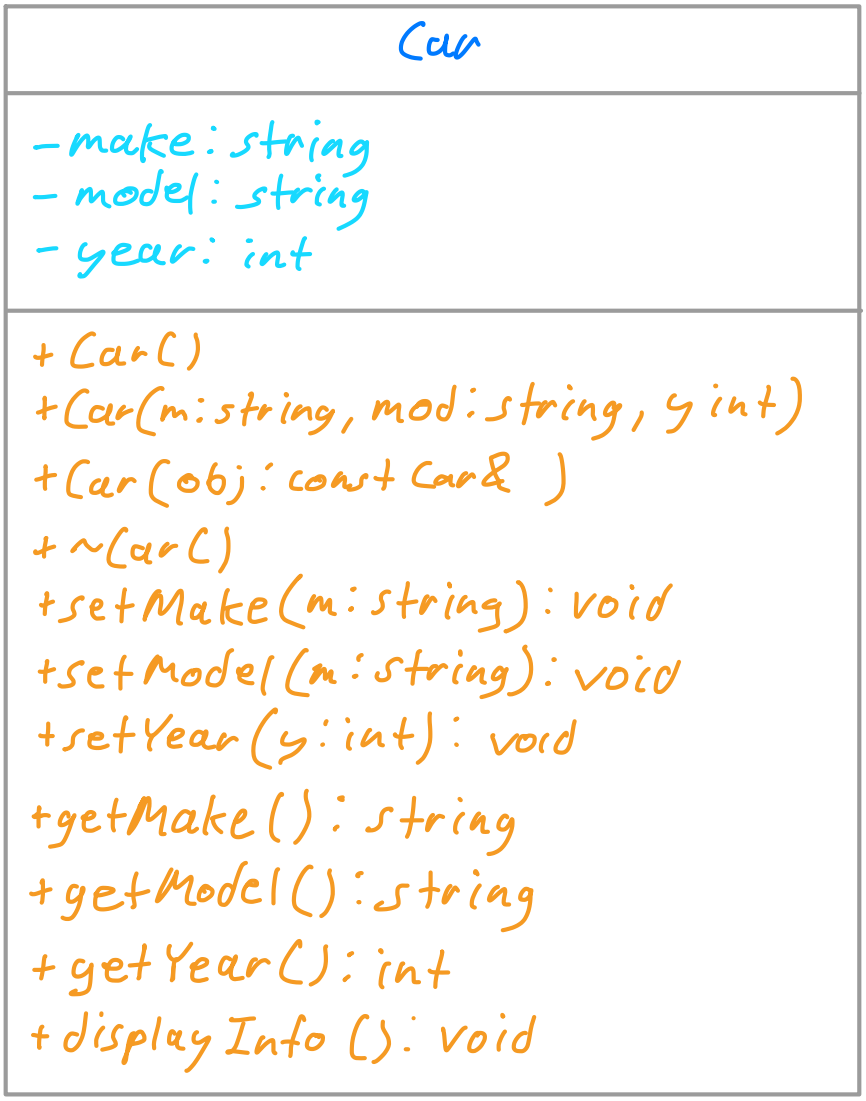
↳ If not char

↳ int x

↳ char *ptr = nullptr;

↳ ptr = reinterpret_cast<char *>(&x)

↳ file.write(reinterpret_cast<char *>(&x), sizeof(x))


Car
- make: string
- model: string
- year: int
+ Car()
+ Car(m: string, mod: string, y int)
+ Car(obj: const Car&)
+ ~Car()
+ setMake(m: string): void
+ setModel(m: string): void
+ setYear(y: int): void
+ getMake(): string
+ getModel(): string
+ getYear(): int
+ displayInfo(): void

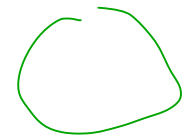
Chapter 13a

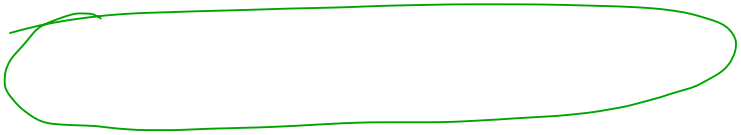
1. Class can only have 1 default constructor (True)
2. A private member function is useful for tasks that are internal to the class but it is not directly called by statements outside the class. (True)
3. Whereas object-oriented programming centers on the object, procedural programming centers on functions. (True)
4. Class objects can be defined prior to the class declaration. (False)
5. A destructor function can have zero to many parameters. (False)
6. In-place member initialization no longer is available in C++11. (False)
7. Examples of access specifiers are private and public.
8. When an object is defined without an argument list for its constructor, the compiler automatically calls the object's default constructor. (True)
9. While a class's member functions may be overloaded, the constructor cannot be overloaded. (False)
10. You must declare all data members of a class before you declare member functions. (False)
11. You can use the technique known as a member initialization list to initialize members of a class. (True)
12. The constructor fxn may not accept arguments (False)
13. Destructors free memory allocated by object (True)
14. When using smart pointers to dynamically allocate objects in C++ 11, it is unnecessary to delete the dynamically allocated objects because the smart pointer will automatically delete them. (True)
15. If you do not declare a destructor function, the compiler will furnish one automatically. (True)
16. More than one constructor may be defined for class (True)
17. You must use the **`private`** access specification for all data members of a class. (False)
18. Constructor functions are often used to allocate memory that will be needed by the object. (True)
19. In object-oriented programming, the object encapsulates both the data and the functions that operate on the data. (True)
20. More than one destructor function may be defined for a class. (False)

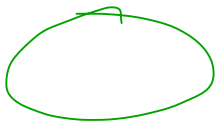
Chapter 7 (1 legit wrong)

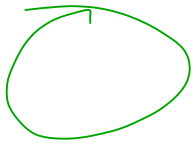
1. Assume `array1` and `array2` are arrays. To assign the contents of `array2` to `array1`, use `array1 = array2;` (False)
2. If you leave out the size declarator in an array definition, you must furnish an initialization list. (True)
3. An array with no elements is illegal in C++. (True)
4. C++ limits the number of array dimensions to two. (False)
5. If an array is partially initialized, the uninitialized elements will be set to zero. (True)
6. An array initialization must be all on one line. (False)
7. To initialize a `vector<int>` named `n` with the values 10 and 20, use `vector<int> n = {10, 20};` (True) glitched→
8. Given `int numbers[] = {99, 87, 66, 55, 101}; cout << numbers[3];`, the output is 55. (True)
9. By using the same subscript, you can build relationships between data stored in two or more arrays. (True)
10. `int scores[25];` is a valid C++ array definition. (True)
11. A `vector` object automatically expands in size to accommodate the items stored in it. (True)
12. An initialization list can be used to specify the starting values of an array. (True)
13. An individual array element can be processed like any other type of C++ variable. (True)
14. A two-dimensional array can be viewed as rows and columns. (True)
15. An array of `string` objects that holds five names is declared as `string names[5];` (True) glitched
16. A statement that correctly defines a `vector` object for holding integers is `vector<int> v;` (True) →
17. When writing functions that accept multi-dimensional arrays, all but the first dimension must be explicitly stated. (True)
18. There is no known way to pass a two-dimensional array to a function. (False)
19. An element of a two-dimensional array is referred to by the row subscript followed by the column subscript. (True)
20. An array's size declarator must be a constant integer expression with a value greater than zero. (True)
21. Given `int scores[] = {83, 62, 77, 97, 86};`, the value 77 is stored in `scores[2];` (True)
22. Arrays must be sized at the time they are declared. (None of these → True statement)
23. In C++11, the range-based `for` loop is best used when you need the element subscript. (False) or executed?
24. When you pass an array as an argument to a function, the function can modify the contents of the array. (True)
25. Each individual element of an array can be accessed by the array name and subscript. (True)
26. An array can store a group of values, but the values must be the same data type. (True)
27. A two-dimensional array of characters can contain strings of the same length, different lengths, or uninitialized elements. (All of these)
28. `int values[1000];` declares an array with 1000 elements. (True)
29. A two-dimensional array can have elements of one data type. (True)
30. An array can easily be stepped through by using a `for` loop. (True)
31. Storing data past an array's boundaries guarantees a compiler error. (False)
32. `int sizes[10];` is a valid C++ array definition. (True)

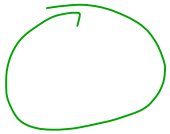
✓ Glitched

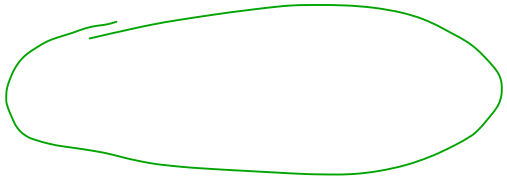


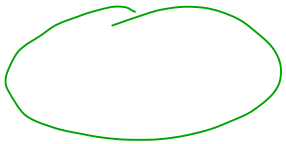


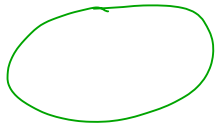


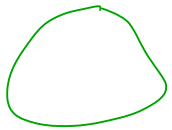




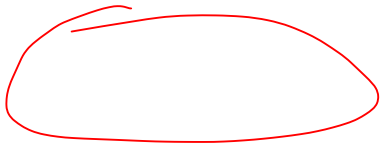




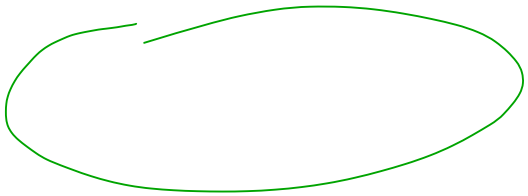


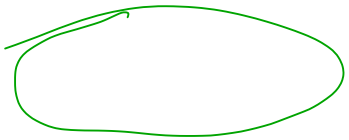


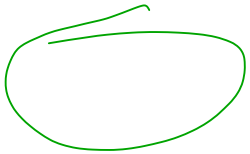
X Question this

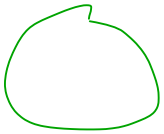


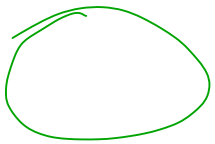

my answer

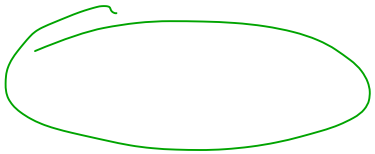




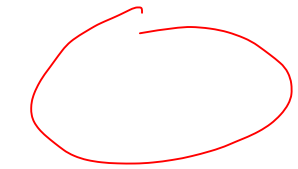


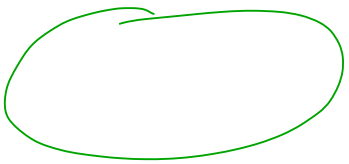






X Question this





← my answer, only fstream can do this