Contents

1 01.21.20 (C++ Ch. 9)

1.1 Encapsulation

- Header files should not contain source code, it should only include prototypes in order to ensure proper information-hiding
- Source code should be placed in a different cpp file, which pulls from the prototypes in the header file

1.2 Include Guards

- Consider the following classes: Student, Course, and Main
  - Student uses Course
  - Main uses Student and Course
  - The main method would then look like:

```
#include "student.h"
#include "course.h"
```

  - student.h compiles properly, but an error is thrown when course.h tries to be included because it has already been included through Student.
  - To fix this, use header guards, as follows:

```
#ifndef FILENAME_H
#define FILENAME_H
```

- Include guards ensure that a prototype is not defined twice
- The header guard should be put in header files that are used in multiple places

1.3 Writing Classes

- Begin by including the necessary header file
- All methods and constructors must be preceded by the header file name and the scope resolution operator (::)

1.4 Constructors & Default Constructors

- Constructors can call other methods and do data-checking
- Constructors can be called explicit with multiple parameters when the parameters are impossible to typecast, as follows:

```
int main () {
  explicit Time t (x = 0, y = 0, z = 0);
} //main
```

2 01.21.20 (C++ Ch. 3)

2.1 Objects and Object Sizes

- An objects size will always be the sum of its data members. The size will not be affected by any methods that are called upon it.
- Because of this, objects can quickly become very large in size.

2.2 UML Diagrams

- Classes are listed as individual boxes
  - top box = class name
  - middle compartment = data members : data type
  - bottom compartment - methods and parameters
    - \- = private
    - \+ = public
    - \# = protected

2.3 Constructors

- Explicit constructors can be used to prevent implicit typecasting, as seen below:

```
class Student {
  Student (int s) {

  } //constructor
} //Student
```

```
int main () {
  Student s {15}; //allowed, completes correctly
  Student c {'C'}; //typecasts automatically, should not occur
  //Note, () can be used in place of {} to construct objects
}
```

- Ex. list initialization with an explicit constructor

```
explicit Account (std::string accountName) //explicit constructor
  : name{accountName} {
  //insert constructor code here
  }
```

3 01.19.20 (C++ Ch. 3)

A look at class creation

```
#include <iostream>
using namespace std;

//defining the class
class GradeBook {
  //holds all public vars, functions
  public:
  //public function
  void displayMessage() {
    cout << "Welcome to your Gradebook" << endl;
  } //displayMesage
} //GradeBook

//main method
int main () {
  //creates a GradeBook object
  GradeBook myGradeBook;
  //calls above created function on object
  myGradeBook.displayMessage();
}
```

- Class functions and vars are, by default, private. The public keyword must be used to denote any public parts of a class.

- Move implementations to a header file for use in main methods while separating out each file.
- When using header files, use quotation marks around them to indicate that they're a file on your machine. Use angle brackets around things to include form the C std lib.
- The purpose of const functions is to prevent the function from modifying the values of data members or objects.

4 01.19.20 (C++ Ch. 2)

A look at some basic C++ code

```
#include <iostream> //enables program to output data

//main function begins program execution
int main () {
  //cout currently a function as a part of the std namespace
  std::cout << "Welcome to C++!\n";
  //above << is an insertion operator, overloaded from the bitwise left-shift

  return 0;
}
```

A look at some higher level C++ code

```
#include <iostream>

int main () {

  int num1{0}; //list initialization
  int num2 = 0; //regular initialization
 //No difference between list & regular initializtion with primitive types.
 //List initialization should be used for UDTs.

  int sum{0}

  std::cin >> num1;
  std::cin >> num2;
```

```
  sum = num1 + num2;

  std::cout << sum << std::endl;
  //endl is helpful because it flushes the buffer
  //newline character does not
  return 0;
}
```

A look at a common mistake

```
#include <iostream>

int main () {
  int x {5};

  if(x > 10); {
    std::cout << x "> 10" << std::endl;
  }
  //still prints output because of semicolon after if statement

  return 0;
}
```