Object Oriented Programming

CMPT 225

Outline

- OOP Basic Principles
- C++ Classes

Examples

- Colours
  - How should we work with colours?
    - How should we store them?
    - How should we modify or operate on them?
- Linked lists
  - How should we provide the functionality of a linked list?
- Shapes
  - …

OOP Principles

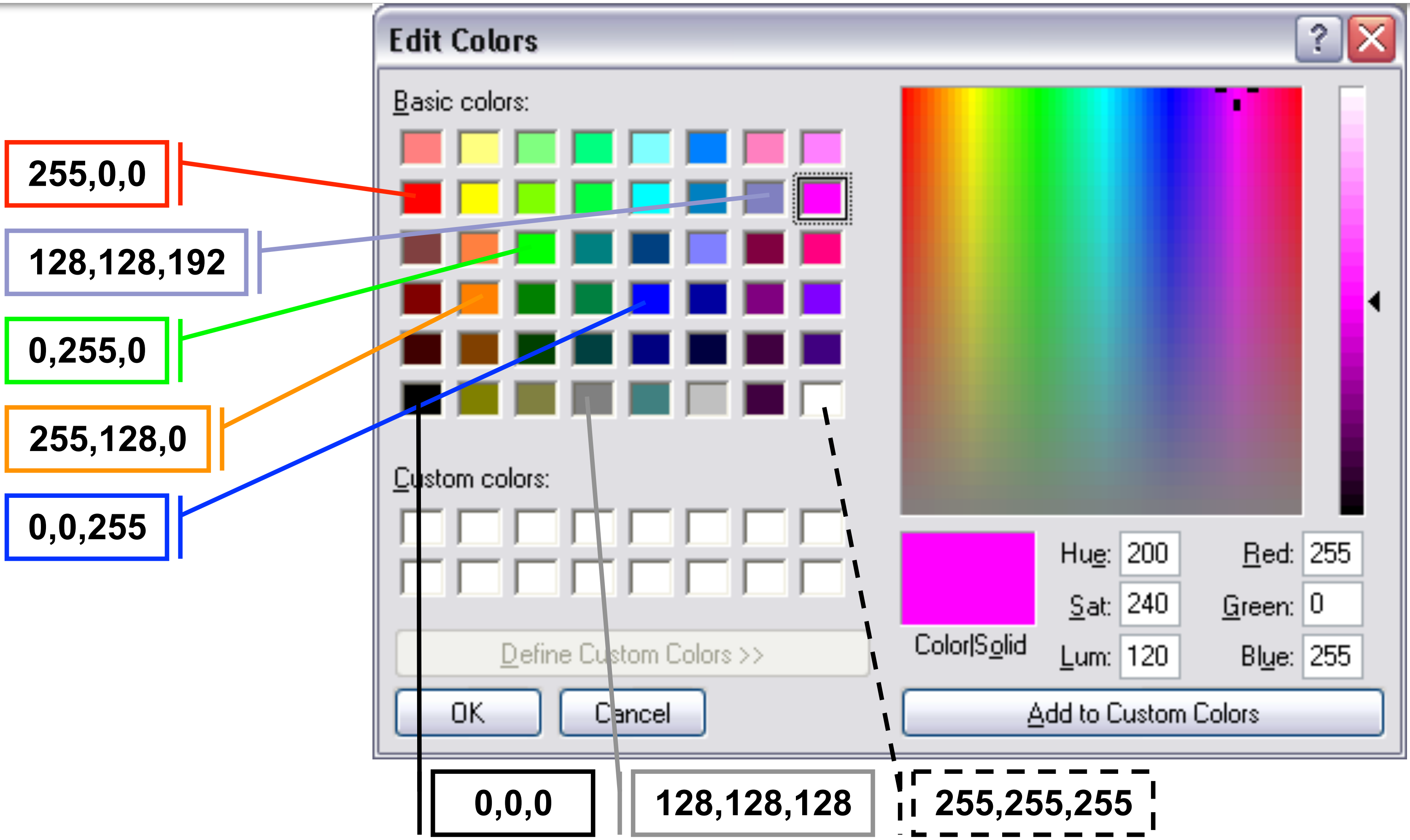
OOP Principles

- Encapsulation
  - Color Class
  - Designing Classes

Representing Colour

- Let's say we need to represent colours
  - There are many different colour models
  - One such is the RGB (red green blue) model
- RGB colours
  - A colour is represented by three numbers, which represent the amount of red, green and blue
  - These values are sometimes recorded as doubles (between 0.0 and 1.0) or sometimes as
  - Integers, between 0 and 255 (or some other number)
    - How many colours can be represented?

Colours and rgb Values



Hue: 200

Sat: 240

Lum: 120

Red: 255

Green: 0

Blue: 255

255,0,0

128,128,192

0,255,0

255,128,0

0,0,255

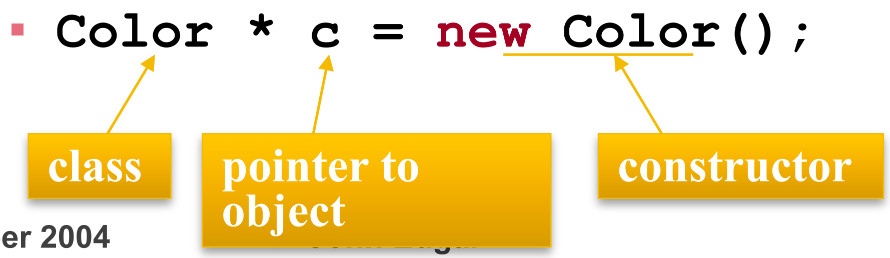
0,0,0

128,128,128

255,255,255

Storing Colour Data

- We need three variables to represent one colour
- It would be convenient to refer to colours in the same way we refer to primitive types
- Object Oriented Programming (OOP) organizes programs to collect variables and methods
  - A **class** is a factory (or blueprint) for creating objects of a particular type
  - An **object** is a collection of variables and methods, and is an **instantiation** of a class
    - `Color * c = new Color();`



class

pointer to object

constructor

Encapsulation

- An object combines both variables and methods in the same construct
  - Variables give the structure of an object
  - Methods dictate its behaviour
  - A class should be a cohesive construct that performs one task (or set of related tasks) well
  - Objects can be used as if they were primitive types
- To encapsulate means to encase or enclose
  - Each object should protect and manage its own information, hiding the inner details
  - Objects should interact with the rest of the system only through a specific set of methods (its public interface)

Classes and Objects

- The class describes the data and operations
    - For colours these include:
        - Attributes for red, green and blue
        - Methods to access and change and create colours
- An individual object is an *instance* of a class
    - Similar to the way that a variable is of a type
    - Each object has its own space in memory, and therefore each object has its own state
        - Individual Color objects represent individual colours, each with their own values for red, green and blue

Information Hiding

- To achieve loose coupling, classes are only allowed to communicate through their interfaces
  - Thereby hiding their implementations details
- Loose coupling is desirable as it:
  - Decreases the chance that changing one module's implementation causes changes to other modules
  - Prevents other modules from assigning invalid values to attributes
- Information hiding is relatively easy to achieve using object oriented programming

Designing a Class

- There are many ways to design classes, as the purpose of classes differs widely
    - Classes may store data and require operations to support this, or
    - May implement an algorithm, or
    - Combine both data and operations
- The initial focus may either be on a class's variables or its methods
- There are, however, some general principles of good design

Design: Make Variables Private

- Variables should generally be made directly inaccessible from outside the class
  - This is achieved by making them **private**
- The values of variables can be accessed using **getter** methods (or **accessor**s)
- New values can be assigned to variables using **setter** methods (or **mutator**s)
  - A setter method assigns the value passed to its parameter to a variable
  - While protecting any class invariants

Design: Write Constructors

- Constructors should initialize all of the variables in an object
- It is often necessary to write more than one constructor
  - Default constructor, with no parameters that assigns default values to variables
  - Constructor with parameters for each variable, that assigns the parameter values to those variables
  - Copy constructor that takes an object of the same class and creates a copy of it

Design: Make Helper Methods Private

- Helper methods are methods that assist class methods in performing their tasks
  - Helper methods are often created to implement part of a complex task or to
  - Perform sub-tasks that are required by more than one class methods
- They are therefore only useful to the class and should not be visible outside the class
  - Helper methods only relate to the implementation of a class, and should not be made part of the interface

Design: Setters Only When Needed

- Class variables are made private
  - To prevent them from being assigned inappropriate values, and
  - To prevent classes from depending on each others' implementations and
- Consider whether or not each variable requires a setter method
  - Is it more appropriate to create a new object rather than changing an existing object's variables?
  - Setters should always respect class invariants

C++ Classes

Basic C++ Classes

- Every C++ class should be divided into header and implementation files
- The header file contains the class definition
- The implementation file contains the definiton of class methods
  - The implementation file has a .cpp extension
  - And should contain the definition of each method declared in the header file
  - Each method name must be preceded by the class name and "::"

C++ Header Files

- The header file has a .h extension and contains
  - Class definition (`class` keyword and class name)
  - Class variables
  - Method declarations (not definitions) for
    - Constructors, a destructor, getters and setters as necessary, and any other methods that are required
  - The class should be divided into public and private sections as necessary

Basic C++ Classes, .h

```
// Thing.h
class Thing
{
public:
    Thing();
    Thing(int startAge);
    //copy constructor and destr
    // made by the compiler
     void display();

private:
    int age; //the one and only attribute
};
```

the file is divided into public and private sections

constructors have the same name as the class and do not have a return type

note the semi-colons

Basic C++ Classes, .cpp

```
// Thing.cpp
#include "thing.h"
#include <iostream>
using namespace std;

Thing::Thing(){
    age = 0;
}//default constructor

Thing::Thing(int startAge){
    age = startAge;
}//constructor

void Thing::display(){
    cout << age << endl;
}//display
```

the file contains method definitions for each method

If a method is not preceded by the class name and :: it is not an implementation of a class method

omitting Thing:: from a method name may not result in a compiler error

C++ Constructors

- If no constructor exists for a class the C++ compiler creates a default constructor
  - Creating **any** constructor prevents this default from being created
- If no copy constructor exists C++ creates one
  - This copy constructor makes a **shallow copy**
    - It only copies the values of data members; which, for pointers, are addresses, and not the dynamically allocated data
    - If the class uses dynamically allocated memory a copy constructor that performs a **deep copy** must be written

C++ Destructors

- Every C++ class must have a **destructor** which is responsible for destroying a class instance
  - `~Thing(); //tilde specifies destructor`
  - A class can have only one destructor
- C++ automatically creates a destructor for a class if one has not been written
  - If a class does not use dynamically allocated memory it can depend on the compiler generated destructor
  - Otherwise a destructor must be written to deallocate any dynamically allocated memory, using delete

Objects in Stack (Static) Memory

- Unlike Java C++ objects do not have to be created in dynamic memory
  - **`Thing th;`** creates a new Thing object in stack memory
    - And calls the default constructor
    - **`Thing th(3);`** would call the second constructor

Copying Objects

Shallow Copies

- Consider a copy constructor for a Linked List

```
LinkedList::LinkedList(LinkedList& ll){
  head = ll.head;
}
```

- This constructor has not created a new list, it has just created a new pointer to the existing list
  - There is still only one list
- This is an example of a **shallow copy**
  - Where only the references are copied, and not the underlying data in dynamic memory

Deep Copies

- A deep copy creates a copy of an object's data and not just its pointers
  - By creating a new object in dynamic memory for each such object in the original
  - For a linked list this would mean traversing the list making a new node for each original node
- Deep copies are required whenever a class allocates space in dynamic memory
  - That is, creates objects using new
- Lab 3 will demonstrate this concept

Summary

Summary

- Object-oriented programming
  - Encapsulation, information hiding
- C++ classes
  - .h file to specify methods/variables, .cpp for details
  - Objects can be created in heap (dynamic) or stack (static) memory

Readings

- Carrano
  - Ch. 8