CSCI 135
Classes

Review of Classes

Some Class Components:

- Name of class
- Class interface: Public operations and [unlikely] data
- Class implementation: Private operations and data

Recall: each instance of a class is an object (similar to the type - variable distinction)

? How and when are data in an object initialized?

! through special **constructor** functions defined with the class

Constructors

- A special kind of member function that is automatically called when object is declared (or allocated using `new`)
- Typically initializes all member variables, but can be used to do anything (*e.g.*, memory management), and importantly to validate data (so that all data is consistent and in range).
- Written like other member functions, but has same name as class and not allowed to have a return value (not even void).

Constructor Example

```
class DayOfYear {
  public:
    DayOfYear(int vmonth, int vday);   constructor prototype
    void output();
    ...
  private:
    int month;
    int day;
};
DayOfYear::DayOfYear(int vmonth, int vday) {
  if ((vmonth > 0) && (vmonth <= 12)) month = vmonth;
  else month = 1;
  day = vday;                          Bad! Should also check
};
int main() {
  DayOfYear date1(11,30);              also calls constructor
  DayOfYear *date2;                    declares a pointer
  date2 = new DayOfYear(11,30);        allocates object w/ 11/30
}
```

Creating, Initializing, and Using Objects

```
class DayOfYear {
  public:
    DayOfYear(int vmonth, int vday);
    void output();
  ...
};
int main() {
  DayOfYear date1(11,30);
  DayOfYear *date2;
  date2 = new DayOfYear(11,30);
  date1.output();          prints 11/30
  date2->output();         note syntax!
  output();                illegal! print which object?
  output(date1);           illegal! doesn't even match prototype
}
```

Way to think: I'm calling a member function foo on an object obj
$\Rightarrow$ `obj.foo(...)`

Re-initializing Objects

An object can be re-initialized by explicitly calling the constructor.

```
int main() {
  DayOfYear date1(11,30);
  DayOfYear *date2;
  date2 = new DayOfYear(11,30);
  ...
  date1 = DayOfYear(12,1);        reinitializing object
  date2 = new DayOfYear(12,1);    works but watch garbage
}
```

Explicit constructor call:

1. Create anonymous object
2. Assigns created object to current object.

Default Constructors

Constructors don't need to have arguments ⇒ called default constructors.

```
class DayOfYear {...}
  public:
    DayOfYear(int vmonth, int vday);
    DayOfYear();                          default constructor
  private:
    ...
};
DayOfYear::DayOfYear() {  // initializes date to Jan 01
  month = 1; day = 1;
};
int main() {
  DayOfYear date2;        No parentheses for default
  DayOfYear date3();      Illegal
  ...
}
```

? What does the parser interpret the date3 line as?

Default Constructors

Constructors don't need to have arguments ⇒ called default constructors.

```
class DayOfYear {...}
  public:
    DayOfYear(int vmonth, int vday);
    DayOfYear();                          default constructor
  private:
    ...
};
DayOfYear::DayOfYear() {  // initializes date to Jan 01
  month = 1; day = 1;
};
int main() {
  DayOfYear date2;        No parentheses for default
  DayOfYear date3();      Illegal
  ...
}
```

? What does the parser interpret the date3 line as?

! A function prototype.

Default Constructor Rules/Guidelines

- Program *should* always define a default constructor
- If no constructors at all are defined, a default constructor is automatically generated if possible (details differ for pre- and post- C++11)
- If no default constructor (either in program, or automatically generated), `SomeClass someObject;` would lead to a syntax error – object can be declared using an available constructor.

Overloading Constructors

Constructors can be overloaded just like other functions (and we did it above).

```
class DayOfYear {
  public:
    DayOfYear(int vmonth, int vday);
    DayOfYear(int vmonth);        overloaded constructor
    DayOfYear();                  default constructor
  private:
    int month;
    int day;
};
DayOfYear::DayOfYear(int vmonth, int vday) {...};
DayOfYear::DayOfYear(int vmonth) {...};
DayOfYear::DayOfYear() {...};
int main() {
  DayOfYear date1(11,30),date2(11),date3;
  ...
}
```

Creates 3 objects, initialized from 3 different constructors.

Another Syntax for Constructor Definition

Instead of:

```
DayOfYear::DayOfYear(int vmonth, int vday)
{
   month = vmonth;
   day = vday;
}
```

We can write this (recall that month and day are private member variables of DayOfYear):

```
DayOfYear::DayOfYear(int vmonth, int vday)
                    : month(vmonth), day(vday)
```

- Most prefer the latter style.

Memory Leaks With Objects

```
class SomeClass {
public:
  SomeClass(...); Constructor, will allocate RAM for parent
  ...
private:
  Node *parent;
  int  data;
}
main() {
  while (...) {
    SomeClass al; Lifetime local to loop
    ...
  }
  ...                    al is dead but al.parent is garbage
```

⚠ We need to avoid the memory leak!

Destructors

- Opposite of constructor
- Used to clean up after object is dead: reclaim all allocated memory, do any processing needed to ensure data integrity, etc.
- Automatically called when object is dead.
- Non-pointer members are automatically destroyed (in reverse order of appearance), and dont need to be handled by destructor.

```
class SomeClass {
public:
  SomeClass(...);
  ~SomeClass();        Destructor
private:
  Node *parent;
  int  data;
};
SomeClass::~SomeClass() {
  delete parent;
  // other clean up activities (e.g., for integrity)
};
```

Class Elements

Recall we can have structs whose fields are structs (*e.g.*, World was a struct which had a Country field). We can have classes that do the same thing!
Ex: a Holiday class which might have different member functions than DayOfYear.

```
class DayOfYear {...};   // same as above
class Holiday {
public:
  Holiday();          default constructor
  Holiday(int month, int day, bool theEnforcement);
  void output();
private:
  DayOfYear date;   member is another object
  bool parkingEnforcement; // true iff enforced
};
Holiday::Holiday() : date(1,1), parkingEnforcement(false);
{};
Holiday::Holiday(int month,int day,bool theEnforcement)
                 : date(month,day),
                   parkingEnforcement(theEnforcement)
{};
```

Example: Object-Oriented Design

Problem Statement:

- Every student has one name and two grades.
- Every grade has an ID (midterm or final), a weight, and a score

Design a Student class with the following capabilities:

1. Input two grades from keyboard
2. Check that weights sum to 1 (exit otherwise)
3. Computes the student's final grade

Example: Object-Oriented Design

Problem Statement:

- Every student has one name and two grades.
- Every grade has an ID (midterm or final), a weight, and a score

Design a Student class with the following capabilities:

1. Input two grades from keyboard
2. Check that weights sum to 1
3. Computes the student's final grade

❓ Step 1: What are the classes?

Example: Object-Oriented Design

Problem Statement:

- Every student has one name and two grades.
- Every grade has an ID (midterm or final), a weight, and a score

Design a Student class with the following capabilities:

1. Input two grades from keyboard
2. Check that weights sum to 1
3. Computes the student's final grade

? Step 1: What are the classes?

! Grade, Student

Example: Object-Oriented Design

Next step: design the classes, keeping in mind ADT principles from before (encapsulation, etc.)

```
enum GradeID {midterm, final};
class Grade {
public:
  GradeID getId();
  void setId(GradeID newid);
  // get/set for all privates?
  void inputGrade();
  void showGrade();
  Grade();
private:
  GradeID id;
  double weight;
  double score;
};
```

```
class Student {
public:
  void setGrade(GradeID id,
                double newWeight,
                double newScore);
  // more gets/sets here
  bool checkWeight();
  double finalGrade();
  Student();
private:
  string name;
  Grade grades[2];
};
```

? What's wrong with above design?

Example: Object-Oriented Design

Next step: design the classes, keeping in mind ADT principles from before (encapsulation, etc.)

```
enum GradeID {midterm, final};
class Grade {
public:
  GradeID getId();
  void setId(GradeID newid);
  // get/set for all privates?
  void inputGrade();
  void showGrade();
  Grade();
private:
  GradeID id;
  double weight;
  double score;
};
```

```
class Student {
public:
  void setGrade(GradeID id,
                double newWeight,
                double newScore);
  // more gets/sets here
  bool checkWeight();
  double finalGrade();
  Student();
private:
  string name;
  Grade grades[2];
};
```

? What's wrong with above design?

! Not encapsulating! Student::setGrade shouldn't need to know about private parts of Grade

Example: Object-Oriented Design

Improving the design:

```
enum GradeID {midterm, final};
class Grade {
public:
  GradeID getId();
  void setId(GradeID newid);
  // get/set for all privates?
  void inputGrade();
  void showGrade();
  Grade();
private:
  GradeID id;
  double weight;
  double score;
};
```

```
class Student {
public:
  void setGrade(Grade newGrade);
  // more gets/sets
  bool checkWeight();
  double finalGrade();
  Student();
private:
  string name;
  Grade grades[2];
};
```

```
bool Student::checkWeight() {
  if (grades[0].getWeight()+grades[1].getWeight() == 1.)
    return true;
  else return false;
};
...
```

Using the previous example

```
int main() {
   Student al;          initialized using constructor
   Grade tempgrade;
   al.setName("al");
   tempgrade.inputGrade();     input al's grades
   al.setGrade(tempgrade);
   if (al.checkWeight() == true)
      cout << al.finalGrade();
   else
      cout << "Weight error in grade\n"
}
```

⚠ We should have written this looking *only* at the class interfaces (and good names help!).

Class vs. Instance Variables

A **class variable** is a variable associated with the class, while an **instance variable** is a variable associated with the object.
$\Rightarrow$ Just one class variable shared by all objects of that class vs. a distinct variable for each object.
Typical uses of class variables:

- Keeping track of how many objects exist
- Keeping track of how many times a member function is called
- Storing constants common to all objects of class

Class Variables in C++

Recall: a variable with static storage class has lifetime that extends beyond its scope, and isn't reallocated each time it enters scope. This is exactly what we want in a class variable!
$\Rightarrow$ C++ approximates class variables through member variables with static storage class.

C++ also generalizes this to static member functions.

Typical use: a function that controls all objects of the class – *e.g.*, deciding whether player A or B has the next turn, when playerA and playerB are objects of the player class.

⚠Static member functions can only use static member data/functions!
Example: Display 7.6 from text, modeling 1 line of clients waiting for service from 2 servers.

Example: Class and Instance Variables

```
class Cube {
public:
  Cube(double width, double height, double depth);
private:
  static int numCubes = 0;      class variable
  double w,h,d;                 instance variables
  const static int maxCubes = 8;
}
Cube::Cube(double width, double height, double depth) {
  if (numCubes < maxCubes) {
    w=width; h=height; d=depth;
    numCubes++;
    }
  else handle error somehow
};
```

? After creating k Cubes (*i.e.*, calling constructor k times), how much memory is used for private variables?

Example: Class and Instance Variables

```
class Cube {
public:
  Cube(double width, double height, double depth);
private:
  static int numCubes = 0;      class variable
  double w,h,d;                 instance variables
  const static int maxCubes = 8;
}
Cube::Cube(double width, double height, double depth) {
  if (numCubes < maxCubes) {
    w=width; h=height; d=depth;
    numCubes++;
    }
  else handle error somehow
};
```

❓ After creating k Cubes (*i.e.*, calling constructor k times), how much memory is used for private variables? ❗ 3k doubles, 2 ints

Nested Classes

```
class OuterClass {
public:
  class InnerClass1 {
  public: ...
  private: ...
  };
private:
  class InnerClass2 {
  public: ...
  private: ...
  };
  int x;
};
OuterClass::InnerClass1 someObject;
```

- Scopes of "InnerClass" identifiers are OuterClass
  $\Rightarrow$ there can be another InnerClass in a different scope.
- InnerClass2 can NOT be used outside of OuterClass (but InnerClass1 can, if referred to as OuterClass::InnerClass1)
- All [public and private] elements of InnerClass* can be accessed by OuterClass member functions

OO Design Steps

1. Understand the problem! For problems involving interaction with users/agents (*i.e.*, most), identify **use cases** that outline the sequence of events for each system action.
2. Identify ADTs (*i.e.*, classes) needed to represent all relevant information for the problem. Don't forget ADTs include data as well as operations. Document ADT interfaces especially well!
3. Determine relationship between classes. Who talks to who? Are any classes special types of other classes (*e.g.*, classA *is-a* classB)?
4. Repeat above steps as many times as needed to improve the design. We still haven't started writing C++ yet!
5. Write C++ code for each class. Since you documented the interface well, others can code code it too.
6. Test your code. Repeat above steps as needed until debugged.

Exercises

1. Finish the Student/Grade example we started
2. Pick any of your 136 labs, and redo it in an object-oriented manner.
3. Do an object-oriented design for an order processing system. This one is wide open!