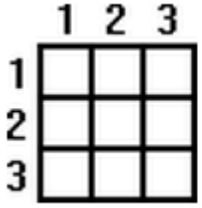
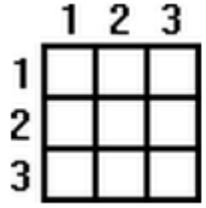
Word Boxes

- Word Boxes are like miniature crossword puzzles, except that each word is filled in across and down the grid.



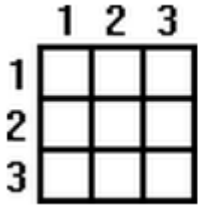
1. bound paper stack
2. what's in the hole
3. study

P	A	D
A	C	E
D	E	N



1. zig counterpart
2. time since birth
3. golly

Z	A	G
A	G	E
G	E	E



1. decay
2. rowboat tool
3. attempt

R	O	T
O	A	R
T	R	Y

```
class DayOfYear {
 public:
  DayOfYear();
  DayOfYear(int month, int day);
  void Output();
  void Input(int month, int day = 1);
 private:
  int month_;
  int day_;
};
```

OOP Design
Separate Compilation
Overloading Rules

Luke Sathrum – CSCI 20

Today's Class

- Object Design Principles
- Structure vs. Class
- Separate Compilation
- Overloading Rules
- Ambiguous Overloading

OOP Design Principles

Design Principles

- With Encapsulation we don’t need to know how the class is programmed
- We just need to know the rules on how to
  - Use
  - Implement
- These rules are known as the interface or API
  - Application Programming Interface

Interface / API

- Consists of two things
  - Comments
    - Tell what the object is supposed to represent
    - Gives an overview of your class
  - Public Member Functions
    - Comments that tell you what the member function does
    - What the function takes as arguments

Interface / API Example

- cplusplus.com has a very good reference with a built-in API
  - http://www.cplusplus.com/reference/
- We'll take a look at the string API
  - http://cplusplus.com/reference/string/string/

Implementation

- Tells how the class interface is realized as C++ code
- Consists of the private and public members of the class
- Basically it is the code you will bring into your program in order to use the interface

Interface and Implementation – Why?

- Allows us to easily give a user just what they need
- Allows us to change the implementation (the backend) and keep the same interface
- Allows multiple people to work on the same project as the interfaces have already been defined

Structure vs. Class

- Technically structures and classes basically do the same thing
  - There are some slight notational differences
  - By default structures are public while classes are private
- We tend to ignore the fact that structures can have member functions
- This allow us to use structures in a different way than classes
  - Use structures to group related data together
  - Use classes to create objects

Summary

- Objects have
  - Interface
  - Implementation
- By keeping these separate it is easier for use to write our code
- Also easier for someone to use our code
- We only use structures for data

Separate Compilation

Separate Compilation

- You can divide your program into parts that are stored in separate files
- Usually we'll separate a class from a main program that implements that class
- When we separate we'll end up with the following
  - An interface file
  - An implementation file
  - A main program file

Interface File

- Called a header file
- Should end with the suffix **`.h`**
- Should be named for the name of your class
  - Should be lowercase
  - **`my_class.h`**
- This interface file should also include all other library functions you will use in your class
  - **`cstdlib`**, **`iostream`**, etc…
- This file holds the declaration of your class

Interface File

- To avoid multiple inclusion of your files we need to have header guards
  - This may happen when more than one file includes your class
- We have a way of telling C++ only include once
  - Your interface file should start with the following
  - `#ifndef LASTNAME_FILENAME_H_`
  - `#define LASTNAME_FILENAME_H_`
- And end with the following
  - `#endif /* LASTNAME_FILENAME_H_ */`
- The interface file should also have header comments for each public member function of your class

Implementation file

- This file holds the definition of your class functions
- Should end with the **`.cpp`** suffix
- Should be named for the name of your class
  - **`my_class.cpp`**
- This file needs to include your class header file
  - **`#include "my_class.h"`**
- Should include header comments for your private member functions

Main Program File

- Include the class header file
  - `#include "my_class.h"`
- Also will need to link your main file to your class files
  - Same process that we do for **`CinReader`**

Summary

- We can break up our interface and implementation
- Interface goes in the .h
- Implementation goes in the .cpp

Sample Code

- Separate Compilation
  - `my_class.h`
  - `my_class.cpp`
  - `separate_compilation.cpp`

Overloading Concepts

Overloading – Which Function?

- How do we know which function the program will use?
- The compiler checks both
  - The number of arguments
  - The types of the arguments

Overloading – Rules

- Functions with the same name must have either
  - Different number of parameters
  - One or more parameters of different types
  - Both
- You **cannot** overload
  - Differ only differ in the return type

```
int F(int num1);
double F(int num1);
```

  - Based on constant (**const**) vs. normal variable

Overloading – Automatic Type Conversion

- Suppose we have the following function:

```
double Car::MPG(double miles, double gallons) {
  return (miles / gallons);
}
```

- And we call it with
  - `my_car.MPG(45, 2);`

Overloading - ATC

```
my_car.MPG(45, 2);
```

- What values are passed to mpg()?
  - 45
  - 2
- What happens when **`MPG()`** receives the values?
  - C++ Automatically converts the integer values to floating-point values

Overloading - ATC

- Automatic Type Conversion can be a problem when we overload functions
- What if we had the following function in addition to the previous function:

```
int Car::MPG(int kilometers, int liters) {
  return (kilometers / liters);
}

double Car::MPG(double miles, double gallons) {
  return (miles / gallons);
}
```

- The Call: **`your_car.MPG(45, 2);`**

Overloading - ATC

- With the same call, **`MPG(45,2)`**, we get the new function even though we are using the same call
- It could be possible that you wanted to compute using the double **`MPG()`** instead of using the other function
- To call the other function you would need to do
  - **`MPG(45.0, 2.0)`**
- You nee to be careful with Automatic Type Conversion

How C++ Resolves Overloading

- Exact Match
  - If the number and types of arguments exactly match a definition then that definition is used
- Match using Automatic Type Conversion
  - If no exact match but there is a match with ATC then that match is used

Ambiguous Overloading

- Sometimes C++ will match more than 1 function
- EX
  - `void MyClass::F(int num1, double num2);`
  - `void MyClass::F(double num1, int num2);`
- The Call
  - `F(98, 99);`
  - Which function gets called?
- Neither… C++ throws an error instead

Summary

- We determine an overloading match based on
  - Number and Types of Parameters
- C++ will automatically type convert for us
- It is possible to write ambiguous overloaded functions

Sample Code

- Ambiguous Overloading
  - `overloading_concepts.cpp`