

## Designing and Using Classes

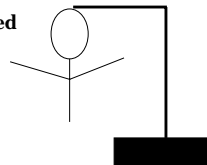
- **Class implementation, summary of what we've seen**
  - Data is private and is accessible in each member function
  - Each object has its own data, so that each of five Dice objects has its own `mySides` and `myRollCount`
  - Member function implementations are in a `.cpp` file, interface is in a `.h` file
- **Compiling and linking, interface and implementation**
  - Client programs `#include` a `.h` file, this is the interface
  - Client programs link the implementation, which is a compiled version of the `.cpp` file (`.o` or `.obj` suffix), implementations are often combined in a library, e.g., `libtapestry`, and the library is linked

## Implementing Classes

- **Determining what classes are needed, and how they should be implemented is difficult; designing functions is difficult**
  - Experience is a good teacher, failure is a good teacher
  - Good design comes from experience, experience comes from bad design*
  - Design and implementation combine into a cyclical process: design, implement, re-visit design, implement, test, redesign, ...
    - Grow a working program, don't do it all at the same time
- **One design methodology says "look for nouns, those are classes", and "look for verbs or scenarios, those are member functions"**
  - Not every noun is a class, not every verb is a method

## Playing Hangman, toward a prototype

- Hangman is a word game, a player tries to guess a secret word one letter at a time, each missed letter counts against the player, after 8 or 10 or 12 misses the player is "hung". Usually each miss results in drawing a body part on a gallows.
  - Diagram shows four misses
  - Part of 10-letter word is guessed
- What are nouns?
- What are verbs?
- What are scenarios?



\_ \_ t \_ \_ a t \_ \_ \_    a m o s n t

## Nouns, scenarios, verbs

- **Get a word to guess**
  - From another player, a dictionary, the web
  - From a *WordSource*
- **Show the word to the player, let the player guess letters**
  - The word is displayed, then letters are revealed as guesses are made
  - Class *Word*, methods *Display*, *Reveal*, *Guess*, ...
- **Guess is also a noun, a letter is guessed, missed letters count against, good letters reveal, duplicate guesses don't count**
  - GussedLetters? Letters? Alphabet? Which is the noun?

## Implementing WordSource

- What's the simplest way to get a word from a WordSource so that we can test the rest of the program
  - Can we design a class that's simple to test with at first, but easy to make more realistic later (essence of prototyping)
  - How can we guess pick one of several words at random once we're ready to move towards a more realistic implementation?
    - Alternatives using small number of strings and a Dice?
    - Alternatives using a file of words?
- What should we do to test the WordSource class?
  - Can we test without implementing the whole program?
  - Test each class separately when possible, isolate mistakes

## wordsource.h, wordsource.cpp

- *WordSource* will return a word, later add "from a file"

```
#include <string>
class WordSource
{
public:
    WordSource();
    string GetWord();
};

// here's the .cpp file
#include "wordsource.h"
WordSource::WordSource()
{
}
string WordSource::GetWord()
{
    return "literature";
}
```

## Guessing letters

- Player guesses a letter, it's in the word, or a miss, or has been guessed already
  - Create a class *Letters*, have it report whether a letter has been guessed already, or a letter is in the word, or a miss
  - Should *Letters* report a miss/correct? If so, does *Letters* need to know the word? What are alternatives?
- Don't worry about implementation, worry about behavior, or the interface
  - Eventually you'll need to worry about implementing, what will be hardest/harder, how can we test without implementing hard part first?

## letters.h

- We'll construct an instance of *Letters* from a secret word/string
  - Ask *Letters* to display the "to be guessed word"
  - Guess a letter, have *Letters* report if it's in the word
  - Optionally report duplicate guesses, add this later

```
class Letters
{
public:
    Letters(const string& s);
    bool GuessLetter(const string& letter);
    void Display();

private:
    string myDisplay; // show this string
    string myString; // the secret word
};
```

## Testing and implementing `letters.cpp`

- `GuessLetter` uses `string::find` to determine miss/correct
  - Must also “save state” so that `Display` shows guesses (and later so that duplicate guess detection works)
  - Initially we can just return true/false to test, no state saved
- We'll test this version, but be thinking about what `Letters::GuessLetter` must do
  - Change state so that display shows guessed letters
  - Ultimately, remember guesses to not penalize twice
  - What about determining when game is over?
  - What about determining # misses so far? Who tracks?

## `hang.cpp`, the main/testing program

```
#include <string>
#include "prompt.h"
#include "letters.h"
#include "wordsource.h"

int main()
{
    WordSource ws;
    string s = ws.GetWord();
    Letters letters(s);
    while (true)
    {
        letters.Display();
        s = PromptString("guess a letter");
        if (letters.GuessLetter(s))
        {
            cout << "that's in the word!!" << endl;
        }
        else
        {
            cout << "that's a miss" << endl;
        }
    }
}
```

## Programming Tips, Heuristics, Help

- Develop a core working program, add to it slowly
  - Iterative enhancement, test as you go, debug as you go
- Do the hard part first, or do the easy part first
  - Which is best? It depends.
- Concentrate on behavior first when designing classes, then on state
  - State is useful for communicating between method calls
- If you're using several classes, you'll need to modify the Makefile or your project in an IDE: Codewarrior/Visual C++

## Common interfaces are a good thing

- The class `WordStreamIterator` iterates over a file returning one word/string at a time

```
string filename = PromptString("enter file name: ");
WordStreamIterator ws;
ws.Open(filename);
for(ws.Init(); ws.HasMore(); ws.Next())
{
    cout << ws.Current() << endl;
}
```
- The class `StringSet` and `StringSetIterator` allow sets of strings to be iterated over one string at a time

```
StringSet sset;
sset.insert("banana"); sset.insert("cherry");
StringSetIterator it(sset);
for(it.Init(); it.HasMore(); it.Next())
{
    cout << it.Current() << endl;
}
```

## Reuse concepts as well as code

- Using the same syntax for iterating saves time in learning about new classes, will save coding when we learn how to exploit the commonality
- We can develop different Question classes and “plug” them into a quiz program if the member functions have the same name
  - See *quiz.cpp*, *mathquest.cpp*, and *capquest.cpp*
  - Programs must #include different headers, and link in different implementations, but quiz.cpp doesn't change
- Random walk classes: one- and two-dimensional, can use the same driver program if the classes use the same method names

## Random walks

- Throwing darts (randomness in programs) is a technique for simulating events/phenomena that would be otherwise difficult
  - Molecular motion is too time-consuming to model exactly, use randomness to approximate behavior
    - Consider the number of molecules in  $10^{-10}$  liters of a gas, each affects the other if we're simulating motion
    - $6.023 \times 10^{23}$  molecules / 22.4 liters is (approx)  $2.7 \times 10^{12}$  molecules
  - If we can do 100 megaflops, what does this mean?
- Simulations are important in many modelling applications, require pseudo-random numbers and some mathematics as well as programming

## Walking behavior (see frogwalk2.cpp)

```
int main()
{
    int numSteps = PromptRange("enter # steps",0,30000);
    RandomWalk frog(numSteps); // define two random walkers
    RandomWalk toad(numSteps);
    int samePadCount = 0; // # times at same location

    frog.Init(); // initialize both walks
    toad.Init();

    while (frog.HasMore() && toad.HasMore())
    {
        if (frog.Current() == toad.Current())
        {
            samePadCount++;
        }
        frog.Next();
        toad.Next();
    }
    cout << "frog position = " << frog.Position() << endl;
    cout << "toad position = " << toad.Position() << endl;
    cout << "# times at same location = " << samePadCount << endl;
    return 0;
}
```

## Two-dimensional walker

- One-d walker Current() returns an int as position
- Two-d walker Current() returns a Point as position
  - Both int and Point can be compared using ==
  - Both int and Point can be printed using <<
- Same program works for two-d walker, even though underneath the implementation is very different
  - Since the interfaces are the same/similar, client programs are easier to write once, use many times
  - Client code still needs to #include a different header and must link in a different (two-d) walker implementation

## What's the Point?

- The two-dimensional walker uses `#include "point.h"`
  - This provides access to class `Point` declaration/interface
  - The class `Point` is actually defined using `struct Point`
  - In C++, a struct is a class in which everything is public by default
    - In a class, everything is private by default
    - A struct is really a hold-over from C, used in C++ for *plain old data*
  - Some programmers/designers don't like to use structs in C++, but use classes only
- We'll use struct when data is public, when the state is really more important than the behavior
  - Guideline, data is private except in a struct, other options?

## point.h

```
struct Point
{
    Point();
    Point(double px, double py);

    string toString() const;
    double distanceFrom(const Point& p) const;
    double x;
    double y;
};
```

- Two constructors, data is public, how is the (0,0) defined?
  - How is distance from (3,5) to (11,20) calculated?
  - How is a `Point p` printed?

## Other details from point.h

- Points can be compared with each other using `==`, `<`, `>=`, etc.
- Point `p` can be printed using `cout << p << endl;`
  - Later we'll learn how to *overload* operators like this
  - For now we'll be clients, using Points like ints, `BigInts`, etc.
- The `struct Point` has constructors and other behavior
  - `distanceFrom` and `toString` constitute the behavior
  - Some programmers think structs shouldn't have any functions, holdover from C rather than C++
- What is implementation of `Point::distanceFrom` like?

## Other uses of structs

- In a program using free (non-class) functions, lots of data is often passed from one function to another
  - In class-based programs data is often, though not always, part of a class and a class object is passed
- Using structs to collect related data makes programs easier to read, modify, and maintain
  - Suppose you want to find mean, mode, and median of the lengths of words in a file, two alternatives:

```
void doFileStats(const string& filename,
                double & mean, int & mode, int & median);

void doFileStats(const string& filename, FileData& data);
```

## More struct conventions

- It's almost always worth including a constructor in a struct

```
struct FileData
{
    FileData()
    {
        myMean = 0.0;
        myMode = 0;
        myMedian = 0;
    }
    double myMean;
    int    myMode;
    int    myMedian;
};
```

- What other data might be included in `FileData`, what about other constructors?

## Class (and struct) conventions

- For debugging and printing it's useful for classes to implement a function `toString()`, that "stringizes" an object
  - Also useful in overloading operator `<<` for an object

```
Point p;
string s = p.toString();
cout << s << " " << p << endl;
```

- When initializing data in a constructor, it's better to use an initializer list than to set values in the constructor body
  - Sometimes initializer lists are required (see next example), so using them at all times leads to more uniform coding that works in more situations

## Initializer lists are sometimes required

- Consider a class that has a private `Dice` data member

```
class Game
{
public:
    Game();
    // more functions
private:
    Dice myDie;
    // more data
};
```

- The instance variable `myDie` must be given a # sides, this cannot be given in the `.h` file/declaration, must be provided in the `.cpp` file/class implementation
  - It's an error if an initializer list isn't use

## Initializer lists

- Here are two versions of an initializer list for `Game::Game()`

```
Game::Game()
: myDie(6)
{ }

// if there's more data, use initializer list
Game::Game()
: myDie(6),
  myName("roulette")
{ }
```

- There can be code in constructor body to do more, e.g., read from a file
  - Sometimes it's useful to call private, helper functions

## Mary Shaw

- Software engineering and software architecture
  - Tools for constructing large software systems
  - Development is a small piece of total cost, maintenance is larger, depends on well-designed and developed techniques
- Interested in computer science, programming, curricula, and canoeing



## Three phases of creating a program

- The preprocessor is a program that processes a file, processing all #include directives (and other preprocessor commands)
  - Takes a file, and creates a *translation unit*
  - Replaces #include “foo.h” with contents of file foo.h, and does this recursively, for all #includes that foo includes and so on
  - Produces input to the next phase of program creation
- The compiler has a translation unit as input and produces compiled object code as output
  - The object code is platform/architecture specific, the source code is (in theory at least) the same on all platforms
  - Some compilers require special treatment, not up to standard C++

## From compiling to linking

- The compilation phase creates an object file, but libraries and other files still need to be linked to create an executable
  - Header files like “dice.h” provide only the interface, enough for the compiler to know that a function call has the right parameters and is used correctly
  - The implementation file, “dice.cpp”, must be compiled and included in the final executable, or the program won’t work (call a dice function, but no one is home?)
- Linking combines object files, some of which may be collected in a library of related files, to create an executable
  - Link the standard library (iostream, for example)
  - Link other libraries depending on program, graphics, tapestry, other application-specific libraries

## Issues in creating a program

- Programming environments create optimized or debug code
  - Use debug version to facilitate development
  - If you need optimization, use it only after a program works
- Some errors are compilation errors, typically language syntax or failure to find a #include'd header file
  - The preprocessor looks in standard places for header files, sometimes this list needs to be changed
- Other errors are linker errors, libraries or object files that are needed aren't included
  - Change programming environment parameters to find the libraries