EE 355 Unit 8

Structs & C++ Classes

Mark Redekopp
OBJECTS
Types and Instances

• A 'type' indicates how much memory will be required, what the bits mean (i.e. data vs. address), and what operations can be performed
  – **int** = 32-bits representing only integer values and supporting +,-,\*,\/,=,==,\,<,\,>, etc.
  – **char** = 32-bit representing an address and supporting * (dereference),\&,+,,- (but not multiply and divide)
  – Types are like **blueprints** for what & how to make a particular 'thing'

• A variable or object is an actual instantiation (allocation of memory) for one of these types
  – int x, double z, char *str;
Abstract Data Types

• Often times we want to represent abstract things (beyond an integer, character, or double)
  – Examples:
    • A pixel, a circle, a student

• Often these abstract types can be represented as a collection of integers, character arrays/strings, etc.
  • A pixel (with R,G,B value)
  • A circle (center_x, center_y, radius)
  • A student (name, ID, major)

• Objects (realized as 'structs' in C and later 'classes' in C++) allow us to aggregate different type variables together to represent a larger 'thing' as well as supporting operations on that 'thing'
  – Can reference the collection with a single name (pixelA, student1)
  – Can access individual components (pixelA.red, student1.id)
Objects

• Objects contain:
  – Data members
    • Data needed to model the object and track its state/operation (just like structs)
  – Methods/Functions
    • Code that operates on the object, modifies it, etc.

• Example: Deck of cards
  – Data members:
    • Array of 52 entries (one for each card) indicating their ordering
    • Top index
  – Methods/Functions
    • Shuffle(), Cut(), Get_top_card()
Structs vs. Classes

- **Structs** (originated in the C language) are the predecessors of **classes** (C++ language)
  - Though **structs** are still valid in C++
- **Classes** form the basis of ‘object-oriented’ programming in the C++ language
- Both are simply a way of aggregating related data together and related **operations** (functions or methods) to model some 'object'
- The majority of the following discussion applies both to **structs** and **classes** equally so pay attention now to make next lecture easier.
Object-Oriented Programming

• Model the application/software as a set of objects that interact with each other

• Objects fuse **data** (i.e. variables) and **functions** (a.k.a methods) that operate on that data into one item (i.e. object)
  – Like structs but now with associated functions/methods

• Objects become the primary method of encapsulation and abstraction
  – **Encapsulation**
    • Hiding of data and implementation details (i.e. make software modular)
    • Only expose a well-defined interface to anyone wanting to use our object
  – **Abstraction**
    • How we decompose the problem and think about our design rather than the actual code
C++ STRINGS
C Strings

• In C, strings are:
  – Character arrays (char mystring[80])
  – Terminated with a NULL character
  – Passed by reference/pointer (char *) to functions
  – Require care when making copies
    • Shallow (only copying the pointer) vs. Deep (copying the entire array of characters)
  – Processed using C String library (<cstring>)
String Function/Library (cstring)

- **int strlen(char *dest)**
- **int strcmp(char *str1, char *str2);**
  - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise
- **char *strcpy(char *dest, char *src);**
  - **strncpy(char *dest, char *src, int n);**
    - Maximum of n characters copied
- **char *strcat(char *dest, char *src);**
  - **strncat(char *dest, char *src, int n);**
    - Maximum of n characters concatenated plus a NULL
- **char *strchr(char *str, char c);**
  - Finds first occurrence of character ‘c’ in str returning a pointer to that character or NULL if the character is not found

In C, we have to pass the C-String as an argument for the function to operate on it.

```c
#include <cstring>
using namespace std;

int main() {
    char temp_buf[5];
    char str[] = "Too much";
    strcpy(temp_buf, str);
    strncpy(temp_buf, str, 4);
    temp_buf[4] = '\0'
    return 0; }
```
Recall our conversation of shallow vs. deep copies

Can we just use the assignment operator, ‘=’ with character arrays?

No, must allocate new storage

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

// store 10 user names of up to 80 chars
// names type is still char **
char *names[10];

int main()
{
    char temp_buf[100];

    cin >> temp_buf; // user enters “Timothy”
    names[0] = temp_buf;

    cin >> temp_buf; // user enters “Christopher”
    names[1] = temp_buf;
    return 0;
}
```
Copy­ing Strings/Character Arrays in C

• No, must allocate new storage

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

// store 10 user names of up to 80 chars
// names type is still char **
char *names[10];

int main()
{
    char temp_buf[100];

    cin >> temp_buf;  // user enters “Timothy”
    names[0] = new char[strlen(temp_buf)+1];
    strcpy(names[0], temp_buf);

    cin >> temp_buf;  // user enters “Christopher”
    names[1] = new char[strlen(temp_buf)+1];
    strcpy(names[1], temp_buf);

    return 0;
}
```
C++ Strings

• So you don't like remembering all these details?
  – You can do it! Don't give up.

• C++ provides a 'string' class that **abstracts** all those worrisome details and **encapsulates** all the code to actually handle:
  – Memory allocation and sizing
  – Deep copy
  – etc.
Object Syntax Overview

• You've already used objects
  – ifstream
  – string

• Can initialize at declaration by passing initial value in ( )
  – Known as a constructor

• Use the dot operator to call an operation (function) on an object or access a data value

• Some special operators can be used on certain object types (+, -, [], etc.) but you have to look them up

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

int main(int argc, char *argv[]) {
    // similar to char s1[] = "CS 103"
    string s1("EE 355");

    // len will have 6
    int len = s1.size();

    // s2 will have "355"
    string s2 = s1.substr(3, 3);

    // s3 will have "EE 355 is fun"
    string s3 = s1 + " is fun";

    // will print 'C'
    cout << s1[0] << endl;
    return 0;
}
```

String and Ifstreams are Examples of Objects

```cpp```
```
String Examples

- **Must:**
  - `#include <string>`
  - `using namespace std;`

- **Initializations / Assignment**
  - Use *initialization constructor*
  - Use `'='` operator
  - Can reassign and all memory allocation will be handled

- **Redefines operators:**
  - `+` (concatenate / append)
  - `+=` (append)
  - `==, !=, >, <, <=, >=` (comparison)
  - `[]` (access individual character)

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

int main(int argc, char * argv[]) {
  int len;
  string s1("EE 355");
  string s2 = "fun";
  s2 = "really fun";
  cout << s1 << " is " << s2 << endl;
  s2 = s2 + "!!!";
  cout << s2 << endl;
  string s3 = s1;
  if (s1 == s3) {
    cout << s1 << " same as " << s3;
    cout << endl;
  }
  cout << "First letter is " << s1[0];
  cout << endl;
}
```

Output:

```
EE 355 is really fun
really fun!!!
EE 355 same as EE 355
First letter is C
```

More String Examples

• Size/Length of string
• Get C String (char *) equiv.
• Find a substring
  – Searches for occurrence of a substring
  – Returns either the index where the substring starts or string::npos
  – std::npos is a constant meaning ‘just beyond the end of the string’...it’s a way of saying ‘Not found’
• Get a substring
  – Pass it the start character and the number of characters to copy
  – Returns a new string
• Others: replace, rfind, etc.

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

int main(int argc, char *argv[]) {
    string s1("abc def");
    cout << "Len of s1: " << s1.size() << endl;

    char my_c_str[80];
    strcpy(my_c_str, s1.c_str());
    cout << my_c_str << endl;

    if(s1.find("bc d") != string::npos)
        cout << "Found bc_d starting at pos=":
        cout << s1.find("bc_d") << endl;

    found = s1.find("def");
    if( found != string::npos){
        string s2 = s1.substr(found,3)
        cout << s2 << endl;
    }
}
```

Output:
Len of s1: 7
The string is: abc def
Found bc_d starting at pos=1
def

http://www.cplusplus.com/reference/string/string/
Exercises

- [http://bits.usc.edu/websheets/?group=cppstrings](http://bits.usc.edu/websheets/?group=cppstrings)
  - Circ_shift
  - Namegame
  - Palindrome
Starting with data...

STRUCTS
Definitions and Instances (Declarations)

- **Objects must first be defined/declared** (as a 'struct' or 'class')
  - The declaration is a blue print that indicates what any instance should look like
  - Identifies the overall name of the struct and its individual component types and names
  - The declaration does not actually create a variable
  - Usually appears outside any function

- **Then any number of instances can be created/instantiated in your code**
  - *Instances* are actual objects created from the definition (blueprint)
  - Declared like other variables

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

// struct definition
struct pixel {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};
// 'pixel' is now a type
// just like 'int' is a type
int main(int argc, char *argv[])
{
    int i,j;
    // instantiations
    pixel pixela;
    pixel image[256][256];
    // make pixela red
    pixela.red = 255;
    pixela.blue = pixela.green = 0;
    // make a green image
    for(i=0; i < 256; i++){
        for(j=0; j < 256; j++){
            image[i][j].green = 255;
            image[i][j].blue = 0;
            image[i][j].red = 0;
        }
    }
    return 0;
}
Membership Operator (.)

- Each variable (and function) in an object definition is called a ‘member’ of the object (i.e. struct or class)
- When declaring an instance/variable of an object, we give the entire object a name, but the individual members are identified with the member names provided in the definition
- We use the . (dot/membership) operator to access that member in an instance of the object
  - Supply the name used in the definition above so that code is in the form:
    \texttt{instance\textunderscore name.member\textunderscore name}

```cpp
#include<stdio.h>
using namespace std;
enum {CS, CECS, EE};
struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[]) {
    int i,j;
    // instantiations
    student my_student;
    // setting values
    strncpy(my_student.name,"Tom Trojan",80);
    my_student.id = 1682942;
    my_student.major = CS;
    if(my_student.major == CECS)
        cout << "You like HW" << endl;
    else
        cout << "You like SW" << endl;
    ...
    return 0;
}
```
Memory View of Objects

• Each instantiation allocates memory for all the members/components of the object (struct or class)

```cpp
#include<iostream>

using namespace std;

struct pixel {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};

int main(int argc, char *argv[]) {
    int i, j;
    // instantiations
    pixel pixela;
    pixel image[256][256];
    ...
    return 0;
}
```
Memory View of Objects

- Objects can have data members that are arrays or even other objects

```cpp
#include<iostream>

using namespace std;

struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[])
{
    int i,j;
    // instantiations
    student s1;
    ...  
    return 0;
}
```
IMPORTANT NOTES ABOUT OBJECTS

Assignment semantics and pointers to objects
Object assignment

• Consider the following initialization of s1

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

enum {CS, CECS};

struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[])
{
    student s1, s2;
    strncpy(s1.name, "Bill", 80);
    s1.id = 5; s1.major = CECS;
    
    // Memory allocation and assignment
    Memory
    0x00 'B' 0x01 'i'
    ... ...
    0x4F 0x00
    0x50 5
    0x54 1
    ...
```
Object assignment

• Assigning one object to another will perform an element by element copy of the source struct to the destination object

```cpp
#include<iostream>
using namespace std;
enum {CS, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[])
{
    student s1,s2;
    strncpy(s1.name,"Bill",80);
    s1.id = 5; s1.major = CECS;
    s2 = s1;
    return 0;
}
```
Pointers to Objects

- We can declare pointers to objects just as any other variable

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

enum {CS, CECS };
struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[]) {
    student s1, *stu_ptr;
    strncpy(s1.name,"Bill",80);
    s1.id = 5; s1.major = CECS;
    stu_ptr = &s1;
    return 0;
}
```
Accessing members from a Pointer

• Can dereference the pointer first then use the dot operator

```
#include<iostream>
using namespace std;
enum {CS, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[]) {
    student s1,*stu_ptr;
    strncpy(s1.name,"Bill",80);
    s1.id = 5; s1.major = CECS;
    stu_ptr = &s1;
    (*stu_ptr).id = 4;
    strncpy( (*stu_ptr).name, "Tom",80);

    return 0;
}
```
Arrow (->) operator

- Save keystrokes & have cleaner looking code by using the arrow (->) operator
  - (*struct_ptr).member equivalent to struct_ptr->member
  - Always of the form: ptr_to_struct->member_name

```cpp
#include<iostream>
using namespace std;
enum {CS, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[]) {
    student s1,*stu_ptr;
    strncpy(s1.name,"Bill",80);
    s1.id = 5; s1.major = CECS;
    stu_ptr = &s1;
    stu_ptr->id = 4;
    strncpy( stu_ptr->name, "Tom",80);
    ...
    return 0;
}
```

Memory:
- 0x100: 'B'
- 0x101: 'i'
- 0x104: 5
- 0x108: 1
- 0x10C: 0
- 0x110: 0
- ... (other memory locations)

Memory:
- 0x100: 'T'
- 0x101: 'o'
- 0x104: 4
- 0x108: 1
- 0x10C: 0
- ... (other memory locations)
Passing Objects as Arguments

- In C, arguments must be a single value [i.e. a single data object / can’t pass an entire array of data, instead pass a pointer]
- Objects are the exception…you can pass an entire struct ‘by value’
  - Will make a copy of the struct and pass it to the function
- Of course, you can always pass a pointer [especially for big objects since pass by value means making a copy of a large objects]

```
#include<iostream>
using namespace std;
struct Point {
    int x;
    int y;
};
void print_point(Point myp)
{
    cout << "(x,y)=" << myp.x << "," << myp.y;
    cout << endl;
}
int main(int argc, char *argv[])
{
    Point p1;
    p1.x = 2; p1.y = 5;
    print_point(p1);
    return 0;
}
```
Returning Objects

- Can only return a single struct from a function [i.e. not an array of objects]
- Will return a copy of the struct indicated — i.e. 'return-by-value'

```cpp
#include<iostream>

using namespace std;
struct Point {
    int x;
    int y;
};
void print_point(Point *myp)
{
    cout << "(x,y)=" << myp->x << "," << myp->y;
    cout << endl;
}
Point make_point()
{
    Point temp;
    temp.x = 3; temp.y = -1;
    return temp;
}
int main(int argc, char *argv[])
{
    Point pl;
    pl = make_point();
    print_point(&pl);
    return 0;
}
```
Object-Oriented Programming

- Model the application/software as a set of objects that interact with each other
- Objects fuse data (i.e. variables) and functions (a.k.a methods) that operate on that data into one item (i.e. object)
  - Like structs but now with associated functions/methods
- Objects become the primary method of encapsulation and abstraction
  - Encapsulation
    - Hiding of data and implementation details (i.e. make software modular)
    - Only expose a well-defined interface to anyone wanting to use our object
  - Abstraction
    - How we decompose the problem and think about our design rather than the actual code
Objects

• Objects contain:
  – Data members
    • Data needed to model the object and track its state/operation (just like structs)
  – Methods/Functions
    • Code that operates on the object, modifies it, etc.

• Example: Deck of cards
  – Data members:
    • Array of 52 entries (one for each card) indicating their ordering
    • Top index
  – Methods/Functions
    • Shuffle(), Cut(), Get_top_card()
C++ Classes

- Classes are the programming construct used to define objects, their data members, and methods/functions.
- Similar idea to structs.
- Steps:
  - Define the class’ data members and function/method prototypes.
  - Write the methods.
  - Instantiate/Declare object variables and use them by calling their methods.
- Terminology:
  - **Class** = Definition/Blueprint of an object.
  - **Object** = Instance of the class, actual allocation of memory, variable, etc.

```
#include <iostream>
using namespaces std;
class Deck {
    public:
        Deck();   // Constructor
        int get_top_card();
    private:
        int cards[52];
        int top_index;
};

// member function implementation
Deck::Deck()
{
    for(int i=0; i < 52; i++)
        cards[i] = i;
}
int Deck::get_top_card()
{
    return cards[top_index++];
}

// Main application
int main(int argc, char *argv[]) {
    Deck d;
    int hand[5];
    d.shuffle();
    d.cut();
    for(int i=0; i < 5; i++){
        hand[i] = d.get_top_card();
    }
    ...
```
C++ Classes

- Classes are the programming construct used to define objects, their data members, and methods/functions.
- Similar idea to structs.
- Steps:
  - Define the class’ data members and function/method prototypes (usually in a separate header file).
  - Write the methods (usually in a separate .cpp file).
  - Instantiate/Declare object variables and use them by calling their methods.
- Terminology:
  - Class = Definition/Blueprint of an object.
  - Object = Instance of the class, actual allocation of memory, variable, etc.

```cpp
class Deck {
public:
  Deck(); // Constructor
  ~Deck(); // Destructor
  void shuffle();
  void cut();
  int get_top_card();
private:
  int cards[52];
  int top_index;
};
```

```cpp
#include<iostream>
#include "deck.h"

// Code for each prototyped method

#include<iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
  Deck d;
  int hand[5];
  d.shuffle();
  d.cut();
  for(int i=0; i < 5; i++){
    hand[i] = d.get_top_card();
  }
}
```
Class Definition

- **class name { ... };**
- Each function or data member can be classified as **public**, **private**, or **protected**
  - These classifications support encapsulation by allowing data/method members to be inaccessible to code that is not a part of the class (i.e. only accessible from within a public class method)
  - Ensure that no other programmer writes code that uses or modifies your object in an unintended way
  - **Private**: Can call or access only by methods/functions that are part of that class
  - **Public**: Can call or access by any other code
  - **Protected**: More on this later
- Everything private by default so you must use “public:” to make things visible
- Make the interface public and the guts/inner-workings private

```cpp
class Deck {
    public:
        Deck();   // Constructor
        ~Deck();  // Destructor
        void shuffle();
        void cut();
        int get_top_card();
    private:
        int cards[52];
        int top_index;
};

#include <iostream>
#include "deck.h"

// Code for each prototyped method

#include <iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
    Deck d;
    int hand[5];
    d.shuffle();
    d.cut();
    d.cards[0] = ACE; // won't compile
    d.top_index = 5; // won't compile
}
```
Constructors / Destructors

- **Constructor** is a function of the same name as the class itself
  - It is called automatically when the object is created (either when declared or when allocated via ‘new’)
  - Use to initialize your object (data members) to desired initial state
  - Returns nothing

- **Destructor** is a function of the same name as class itself with a ‘~’ in front
  - Called automatically when object goes out of scope (i.e. when it is deallocated by ‘delete’ or when scope completes)
  - Use to free/delete any memory allocated by the object
  - Returns nothing
  - [Note: Currently we do not have occasion to destructors; we will see reasons later on in the course]

```cpp
class Deck {
    public:
        Deck();   // Constructor
        ~Deck();  // Destructor
        ...
};

#include<iostream>
#include “deck.h”

Deck::Deck() {
        top_index = 0;
        for(int i=0; i < 52; i++){
            cards[i] = i;
        }
}

Deck::~Deck() {
}

#include<iostream>
#include “deck.h”

int main(int argc, char *argv[]) {
    Deck d; // Deck() is called
    ...
    return 1;
    // ~Deck() is called since // function is done
}
```
Writing Member Functions

- When writing member functions, the compiler somehow needs to know that the function is a member of a particular class and that the function has inherent access to data members (w/o declaring them). Thus we must ‘scope’ our functions
- Include the name of the class followed by ‘::’ just before name of function
- This allows the compiler to check access to private/public variables
  - Without the scope operator [i.e. void shuffle() rather than void Deck::shuffle()] the compiler would think that the function is some outside function (not a member of Deck) and thus generate an error when it tried to access the data members (i.e. cards array and top_index).
Calling Member Functions

- Member functions are called by preceding their name with the specific object that it should operate on.
- `d1.shuffle()` indicates the code of `shuffle()` should be operating implicitly on `d1`’s data member vs. `d2` or any other Deck object.

```cpp
#include <iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
    Deck d1, d2;
    int hand[5];
    d1.shuffle();
    // not Deck.shuffle() or shuffle(d1), etc.
    for (int i = 0; i < 5; i++) {
        hand[i] = d1.get_top_card();
    }
}
```
Calling Member Functions

- Within a member function we can just call other member functions directly.

```cpp
#include <iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
    Deck d1, d2;
    int hand[5];
    d1.shuffle();
    ...
}

#include <iostream>
#include "deck.h"

void Deck::shuffle()
{
    cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++){
        int r = rand() % (52-i);
        int temp = cards[r];
        cards[r] = cards[i];
        cards[i] = temp;
    }
}

void Deck::cut()
{
    // swap 1st half of deck w/ 2nd
}
```

D1’s data will be modified (shuffled and cut)

D1 is implicitly passed to shuffle()

Since shuffle was implicitly working on D1’s data, D1 is again implicitly passed to cut()
Exercises

• http://bits.usc.edu/websheets/?group=classes
  – Clicker
  – Recorder
Class Pointers

- Can declare pointers to these new class types
- Use ‘->’ operator to access member functions or data

```cpp
#include<iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
    Deck *d1;
    int hand[5];
    d1 = new Deck;
    d1->shuffle();
    for(int i=0; i < 5; i++){
        hand[i] = d1->get_top_card();
    }
}
```
Multiple Constructors

- Can have multiple constructors with different argument lists

```cpp
#include<iostream>
#include "deck.h"

int main()
{
    Student s1; // calls Constructor 1
    string myname;
    cin >> myname;
    s1.set_name(myname);
    s1.set_id(214952);
    s1.set_gpa(3.67);

    Student s2(myname, 32421, 4.0); // calls Constructor 2
}
```

```cpp
class Student {
public:
    Student(); // Constructor 1
    Student(string name, int id, double gpa);  // Constructor 2
    ~Student(); // Destructor
    string get_name();
    int get_id();
    double get_gpa();
    void set_name(string name);
    void set_id(int id);
    void set_gpa(double gpa);
private:
    string _name;
    int _id;
    double _gpa;
};
```
Public / Private and Structs vs. Classes

• In C++ the only difference between structs and classes is structs default to public access, classes default to private access

• Thus, other code (non-member functions of the class) cannot access private class members directly

```cpp
class Student {
public:
    Student(); // Constructor 1
    Student(string name, int id, double gpa); // Constructor 2
    ~Student(); // Destructor
    ...

private:
    string _name;
    int _id;
    double _gpa;
};
```

```cpp
#include<iostream>
#include "student.h"

int main() {
    Student s1; string myname;
    cin >> myname;
    s1._name = myname; //compile error
    ...
}
```
Accessor / Mutator Methods

- Define public “get” (accessor) and “set” (mutator) functions to let other code access desired private data members
- Use 'const' after argument list for accessor methods

```cpp
#include<iostream>
#include "deck.h"

int main()
{
    Student s1; string myname;
    cin >> myname;
    s1.set_name(mynname);
    string another_name;
    another_name = s1.get_name();
    ...
}
```

```cpp
class Student {
public:
    Student();  // Constructor 1
Student(string name, int id, double gpa);
    // Constructor 2
~Student();  // Destructor
    string get_name() const;
    int get_id() const;
    double get_gpa() const;
    void set_name(string s);
    void set_id(int i);
    void set_gpa(double g);

private:
    string _name;
    int _id;
    double _gpa;
};
```

```cpp
string Student::get_name()
{
    return _name;
}
int Student::get_id()
{
    return _id;
}
void Student::set_name(string s)
{
    _name = s;
}
void Student::set_gpa(double g)
{
    _gpa = g;
}
```
Class Example: Basic Clock

• What members does it need?
• What operations might it have?
• How would we test it?
• Let's write some code...
Class Example: Basic Clock

• What members does it need?
  – int hours, int minutes, int seconds

• What operations might it have?
  – get/set time
  – Print the time
  – Increment hours, minutes or seconds
  – Compare two clock times for equality

• Let's write some code...
C++ Classes: Example

```
#include "clock.h"

Clock::Clock()
{
    hours = 12; minutes = 0;
    seconds = 0;
}

Clock::Clock(int h, int m, int s)
{
    hours = h; minutes = m;
    seconds = s;
}

void Clock::print() const
{
    cout << hours << ":" ;
    cout << minutes << ":" ;
    cout << seconds;
}
```

```
#ifndef CLOCK_H
#define CLOCK_H

class Clock
{
    private:
        int hours;
        int minutes;
        int seconds;
    public:
        Clock();
        Clock(int h, int m, int s);
        void print() const;
};
#endif
```

```
clock.h
```

```
clock.cpp
```
UML (Unified Modeling Language)

• Shows class definitions in a language-agnostic way
• Shows class hierarchy (inheritance, etc.)
• Each class shown in one box with 3 sections
  – Class Name, Member functions, then Data members
  – Precede function/data member with:
    + (public), - (private), # (protected)
  – Functions show name with arguments : return type
  – Data members show name : type

```cpp
class Deck {
    public:
        Deck();   // Constructor
        ~Deck();  // Destructor
        void shuffle();
        void cut();
        int getTop();

    private:
        int cards[52];
        int top_index;
};
```
this Pointer

• How do member functions know which object’s data to be operating on?
• D1 is implicitly passed via a special pointer call the ‘this’ pointer

```
#include<iostream>
#include “deck.h”

int main(int argc, char *argv[]) {
    Deck d1, d2;
    int hand[5];
    d1.shuffle();
}
```

```
#include<iostream>
#include “deck.h”

void Deck::shuffle()
{
    cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++){
        int r = rand() % (52-i);
        int temp = cards[r];
        cards[r] = cards[i];
        cards[i] = temp;
    }
}
```

```
#include<iostream>
#include “deck.h”

void Deck::shuffle(Deck *this)
{
    this->cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++){
        int r = rand() % (52-i);
        int temp = this->cards[r];
        this->cards[r] = cards[i];
        this->cards[i] = temp;
    }
}
```